PIE Developers

1.1, November, 1993

This is a reprint of the premier issue of *PIE Developers*. The cover's artwork has been removed because of a potential infringement on an Apple trademark. Also, this reprint is in black and white, the original was color. The real magazine looks much better than this, honest! We hope you like it.

Editor's Note

PIE Crust

It's tough to decide how to start the editor's note of a brand new developer publication. How about: welcome, and thanks for taking the time to give us a try.

With *PIE Developers* we intend to focus on providing developers with in-depth technical information about the complete range of technologies originating from Apple Computer's Personal Interactive Electronics (PIE) division. The difficulty is that at any point in time there are beginning, intermediate, and advanced developers. We are going to try and provide the right information, at the right depth, for all three audiences.

More importantly, however, is how we want to do this–with a high degree of customer satisfaction. We want every reader to feel as though he or she has found at a minimum one very useful piece of information in ever issue, a piece of information that can save them a few hours of coding, debugging, or research. If we accomplish that goal than we have succeeded.

To that end we've included with this first issue a reader survey. Please take a few minutes to fill out the survey and return it to us. We want to know exactly what you like and don't like about this issue so that we can make subsequent issues even better.

In this first issue, for beginning PIE programmers, we have Howard Oakley's overview of Newton programming. Both Howard and Jim Lanford answer some of the most frequently-asked questions about hardware, software, and programming skills. To reinforce this introductory information, we have code walkthroughs of not one, but two distinct calculator applications by Howard Oakley and Mac Foster. Incidentally, the complete source code for both of these programs, plus a lot more, can be found on the *PIE Developers* source code disk (see the inside front cover for details).

For those of you that have had some experience programming a Newton, we have two short articles about specific techniques. George Penne explains how to download ASCII data from your Mac and convert it to Unicode data on your Newton. Mike Engber of PIE tech support at Apple explains how to include the "smoke and poof" scrubbing response in your code.

Finally, for somewhat advanced Newton programmers we have two technical features by Mike Engber. One describes at length how to work with union-soups. The other details everything you ever wanted to know (and probably couldn't find anywhere else) about views. These are the definitive publishings on these two topics. To round out the issue, we have a distinct selection of tech support threads culled from the Internet by Robert Bruce, keeper of the official Newton archive at Johns Hopkins University.

We think we've rounded up a pretty good batch of articles. We won't declare victory, however, until we receive your surveys. π

A PIE without a crust is just a mound of filling.

PIE Developers™

PIE Developrs is always interested in articles by qualified PIE programmers. Please direct inquiries to Creative Digital Systems.

is copyright © 1993 and published bimonthly by

Creative Digital Systems
293 Corbett Avenue
San Francisco, CA 94114
415.621.4252
415.621.4922 (fax)

cds.sem@applelink.apple.com
72722.3225@compuserve.com
cds sem@aol.com

A One-year subscription to PIE Developers costs $60 in the US and Canada, and $75 elsewhere.

Editor and Publisher — *Steve Mann*
Contributing Editor — *Howard Oakley*
Editorial Advisors — *Kent Sandvik*, *Bob Foster*

Real Miracles Sensibly Priced

Bob Foster
Object Factory Incorporated
objfactory@applelink.apple.com

On September 28, Apple quietly announced it was revamping its developer support policies and, by the way, announced its $2850-a-year PIE Partner program for Newton developers. Developers had been warned of this event the week before by *MacWeek*, and had used the notice to good advantage burning up the wires on CompuServe and Internet bemoaning, denouncing, grimly forecasting, and saying those things developers say when asked to pay for products and services. So, predictably, by the time Apple actually announced its programs, the storm had by and large blown over. Most small or part-time developers realized that they wouldn't need this premium service to develop for the Newton—any more than they had needed Apple's Macintosh Partners program in the past. Most larger developers realized that $3000 is about the cost of sending one developer to Newton school for a week, if you take PIE Partners to be an R&D program, or one full-page ad in a computer magazine, if you take it to be a marketing program.

If Apple charges for debugging a developer's code, the developer may at least console itself that Apple doesn't charge nearly as much as plumbers do for similar services. In fact, we feel that this aspect of PIE Partners is underpriced. We wouldn't agree to help debug all of any company's code for $2850 a year. (We would, however, be willing to call a company "strategic" for considerably less—say half—if there is any interest in this service, unbundled.)

The reason the cost of PIE Partners is a non-issue is that almost everyone who thinks it is worth any extra cost at all is able to afford it, and anyone who doesn't is able to get perfectly adequate support without it. Small Macintosh developers have always found good value in Apple's Associates program; documentation is provided, questions of fact get answered, developers debug their own code. Apple can provide PIE Associate service at a reasonable cost and charge a modest price because the labor is spread over many developers, and not incidentally because Apple makes money from the AppleLink charges developers incur taking advantage of the service. Developers on a truly tight budget have always been able to get reasonable peer support from the likes of CompuServe, AOL, and Internet; all signs are this will continue be true for Newton.

even as the gouge sinks in, one must admire the skill of the carpenter

Still, the Newton has been a rolling wake-up call for developers. No sooner does the developer push the snooze button than Apple's cash register rings again. The $795 price for the Newton Toolkit. The new support pricing. The 1% surcharge on net income. If there is any evidence of Nintendo-like developer gouging, it will be found in the last item. Yet, even as the gouge sinks in, one must admire the skill of the carpenter.

In imposing the NTK tax, Apple has avoided the shaky ground of licensing its technology as embedded in the MessagePad at one price for end users and another for developers, and gone for the time-honored practice of licensing the result of using its development tool. The runtime component which usually justifies such a

Guest Editorial

royalty is almost negligible in the case of the NTK, but we are sure Apple's lawyers can rationalize the charge. By setting the royalty level at 1%, Apple has seemingly reduced the pain—and the incentive for finding a path around the NTK—to a level that will only be felt by the largest customers.

Moreover, the NTK itself is a well-conceived tool, with only minor awkwardness (such as having to manually Apply each change to ensure it is not discarded, or the awesomely slow load times), showing great promise. Due as much to NewtonScript as to the NTK itself, it is already, in beta, at par with development environments available for the Macintosh.

At the same time, assuming the Newton developer market is worth the effort, the 1% tariff seems to open the door to competition. Apple's chief advantage, at this point, is timing and secrecy. But time has a way of passing, and it would seem naive of Apple to suppose it can keep others from learning how to download an application to the Newton. A battle between the NTK and a competing tool that cost the same but charged no royalty would be interesting to watch. A complaint of tool developers for the Macintosh has always been that Apple prices its own tools, like ResEdit, so low that it kills the market for third parties. Now that Apple has set a profitable price level, we can't help but wonder if someone else won't jump in? π

Newton Development–Getting Started

Glimpse of a New Era

Howard Oakley
uk0392@applelink.apple.com

With near-pandemonium at its launch in Boston, and reactions ranging from the avatarik to the skeptical, the arrival of the first of Apple's Newtons has been anything but dull. Surprisingly those previously laying siege to the Macintosh System with calls for pre-emptive multitasking, thorough use of objects, official class libraries, and so on, have yet to realize how many of their demands have been met in the Newton operating system.

A few thousand words cannot hope to provide the most superficial of introductions to Newton development, nor should it pretend to do so when both the MessagePad and the Newton Toolkit have been shipping for only a few months. Instead, I shall attempt to provide you with an impressionist tour of some of the more interesting features.

The Newton

The hardware heart of the Newton MessagePad is an ARM610 RISC processor, with a built-in memory management unit (MMU), 4 Mbyte ROM, and 640 Kbyte RAM (of which about 190 Kbyte is free for the user). The ARM processor, designed in the UK by an Apple joint venture company, supports only 11 basic instructions, which can be variously modified by condition fields, link bits, data processing flags, and the like. Peak performance at a clock speed of 20 MHz is 20 MIPS, with 15 MIPS sustainable. The Apple-specified MMU translates virtual addresses into physical ones, and controls memory access permissions. Memory access can be based on Sections (1 Mbyte blocks) or Pages (Small Pages of 4 Kbyte size with 1 Kbyte Sub-Pages, or Large Pages of 64 Kbyte size with 16 Kbyte Sub-Pages). A Translation Look-aside Buffer supports cached look-ups for most memory accesses in practice.

The sheer size of the Newton ROM suggests that there is extraordinary capability, and early indications are that this is so. Not only does the ROM contain prototype windows and views for the whole of the human interface, but it also contains support for interactive digital books, such as the inclusion of styled text and PICT illustrations, browsing, bookmarks, addition of annotations by the user, and so on.

A hand-portable or palmtop computer poses its greatest problems in two areas–battery power, and user control. In order to keep size and weight to a minimum, Apple has opted in this first Newton for 4 x AAA batteries (or their equivalent NiCad pack). This militates against systems which have hard disks, large quantities of memory, and the other trammels of modern desktop computers. A large number of design features, if not most of the architecture of the Newton, addresses matters originating here.

The human interface, or how a person armed with just a plastic stylus can drive a sophisticated computer system without months of training, is another key consideration. While there are other options without resorting to 'toy' keyboards and mouse-substitutes, a great deal of effort has been devoted to the development of gestural control. Although in some respects these gestures can be tricky to acquire (inserting a line end in Notes, for example), they are skills which users are likely to learn with time and practice. Human interface guidelines and codification of such gestures are in preparation.

Look no files

The Newton ROM lacks a conventional file system, or even any clear distinction between RAM and memory card storage. Instead, it sees storage devices as 'stores' regardless of their nature and location. With a single PCMCIA Type II Version 2.0 card slot, the MessagePad knows of its internal store (RAM) and an inserted card, and allows user-controlled items such as applications and data to be kept on either.

Rather than structure data as documents, applications create and manipulate 'soups' which are indexed pools of entries, where each entry is a structured record. Soups are accessible to all other applications, encouraging add-ons which enhance existing programs, such as indexing notepad entries, or building time and cost summaries from the Dates soup. Soups are a superb implementation of persistent object storage. Once an application creates a soup entry, there is no intermediate step to save it–the object persists until it is removed from the soup.

NewtonScript

NewtonScript is neither Dylan nor a relative of other AppleScript languages, but a relative of SELF, an object-oriented dynamic language (OODL) developed originally by Smith and Unger at Stanford in the mid-80s. Significant contributions to SELF were made by Chambers and others, in a series of projects influenced by work at Xerox PARC (where Randall Smith worked before moving to Sun), and funded by Xerox, Sun, Apple, and others. SELF is remarkable among OODLs for eschewing classes in favor of prototypes. Designed primarily for exploratory programming, SELF has an idiosyncratic syntax which NewtonScript has wisely avoided.

Although in SELF 'everything is an object', NewtonScript has two kinds of data references: immediate references (which include 30-bit integers, Unicode characters, and the Booleans TRUE and NIL) and pointer references. The latter include all data types which are stored as objects in the heap, such as symbols, real numbers, arrays (which can include mixed data types), strings (arrays of characters), and frames (record or struct-like collections of named references of any data type).

Frames and Objects

Named references within a frame are called slots. An example frame declaration is:

```
demoFrame := { // everything between the curly
               // brackets is the frame contents
  index: 1,    // an integer constant slot
  name: "The First",  // a string constant slot
  anotherFrame:       // a frame in a frame
   { date: "1 Sep 93",
     time: "23:07",
     country: "UK" },
  emptySlot: nil,     // slot with nil pointer
  aFunction:          // a function
   func(param)
   begin
     return param * (3 + param);
   end
  };
```

A frame is an object and NewtonScript objects are frames. In a classless OODL, demoFrame is a unique object which has no parent objects. It can be used as a prototype for a child object:

With near-pandemonium at its launch in Boston, and reactions ranging from the avatarik to the skeptical, the arrival of the first of Apple's Newtons has been anything but dull.

```
demoChild := { // slot reference to proto object
  _proto: demoFrame,
  dataArray:
   [                  // an array slot
   series: [1, 2, 3], // an array w/in an array
   name: "OneToThree"
   {                  // a frame within an array
     slotA: 25,
     slotB: 30,
     slotC: "five"
   } ],
  emptySlot: [2, 4, 6] // overridden slot
                       // from prototype
  };
```

demoChild inherits all the slots of the prototype object, demoFrame, except for emptySlot, which is overridden. demoChild also has an additional slot, dataArray.

Inheritance from a prototype is achieved by reference and not by duplication. Thus, if the name slot in demoFrame (also known as demoFrame.name) will be altered within demoChild, it should be declared as an overridden slot for demoChild. If this is not done, and

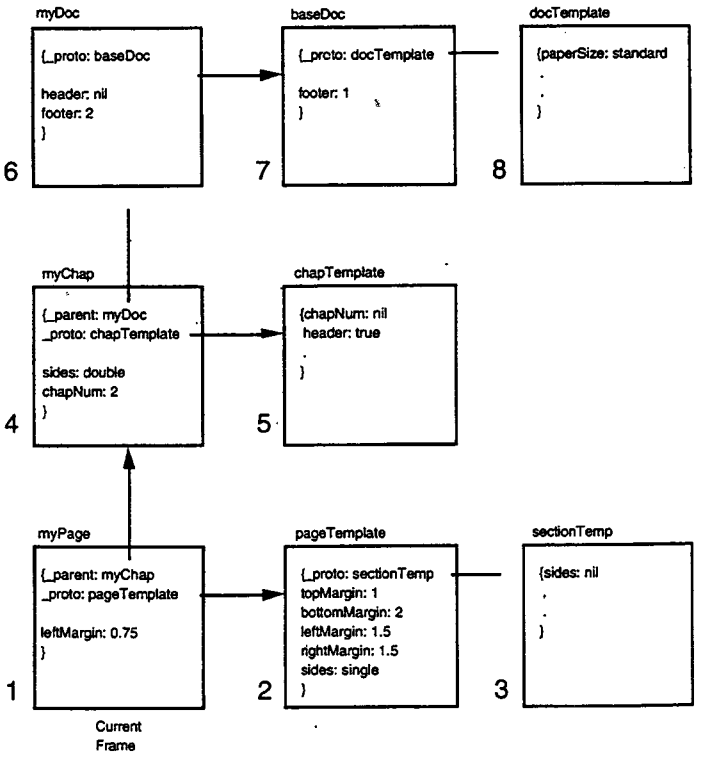


Figure 1 - Prototype and parent inheritance interaction order. This diagram © copyright 1993 by Apple Computer.

demoFrame is in a read-only store such as ROM, `demoChild.name` refers back to `demoFrame.name`, a read-only object. A common NewtonScript error is to attempt to write to slots which are not overridden, and the prototype is in ROM. For view templates this is doomed to failure and usually creates obscure bugs. The answer is to clone from the template, or to create the view at runtime, or simply to add the slot to the view during editing in the NTK.

Double Inheritance

Whilst the most recent versions of SELF offer multiple inheritance with an elaborate prioritization scheme to resolve ordered and unordered tiebreaking, the initial NewtonScript release permits a maximum of two directly ancestral prototypes to an object. These are differentiated as the prototype (identified in the _proto slot of the child object) and the parent (in the _parent slot). Slot conflicts are resolved by the distinct behavior of the two types of inheritance.

When accessing a slot from a given frame object, if it is not further specialized, the first frame searched is the current frame object. If the slot is not found there, then any prototype frame identified in the _proto slot of that frame is searched next. Once the prototype chain has been searched, any parent frame identified in the _parent slot of the first frame is searched, followed by its prototypes. This is illustrated in Figure 1. Note that if no slot is found, then an error is generated, typically an interpreter error in the range -48807 to -48809.

Setting slot values also differs between prototypes and parents. If a slot is found in the current frame, then it is set there, otherwise the search passes to the prototype of the current frame. Although the NTK manual states that "if the slot exists [in] the prototype chain of the current frame, it is copied into the current frame and its value is set there", it is probably bad practice to rely on this (and it does not always appear to work this way). Search then passes to the parent of the current frame, and the slot is set there if found. The search can continue in the prototype chain of the parent using the same rules as in the prototype of the current frame.

Slot search can be specialized at any time by prepending a frame search path to a slot's reference. For instance, the overridden slot in demoChild could be accessed using `demoFrame.emptySlot`. Access may also take place using longer paths, such as `self._parent.theSlot`. In these cases, search beyond the cited path uses only prototypes–parents are excluded unless the path specifically includes them. Also, calling an inherited method such as `inherited:viewClickScript()` only searches the prototype chain and not parents. The more specific that you make a search path, the greater the performance gain.

The keyword self is superficially similar to its namesake in Object Pascal. It normally refers to the current frame. However, self is actually more complex–it refers to the last frame which received a message. Although this is normally the currently executing frame, this is not necessarily so, and misdirection can occur. This is particularly important in using apply() and call(), which both set self to {}. The global function Perform(frame, message, parameterArray) can also be used to avoid misdirection.

Control structures

NewtonScript contains traditional constructs for controlling program flow, including the following:

```
if ... then ... else ...
for ... to ... by ... do ...
loop ... break ...
while ... do ...
repeat ... until ....
```

Currently, there is no case statement. An innovative iterator, which let you iterate over the elements of arrays and frames, is foreach. This is demonstrated in the following code fragments:

```
foreach slot, value in aFrame do
  begin
    Print(slot);   // print slot name to debugger
    Print(value);  // print out slot's contents
  end;

foreach index, value in anArray do
  begin
    Print(index);  // print index to debugger
    Print(value);  // print out contents
  end;
```

Because NewtonScript is a dynamic language, where variable length lists can be represented as arrays, the latter is an effective list iterator.

Exception handling

One of the most sophisticated proposals in the Dylan language description is a specific set of features to support exception handling. Although NewtonScript's exceptional handling is not as sophisticated, it is still robust, as illustrated in the following code:

Soups are a superb implementation of persistent object storage.

```
begin
  local myErr, answer;
  myErr := 0;
  try
   begin
   // do various things
   if IsNan(answer) then
    begin
      answer := 0.0;
      myErr := -48404;
      throw('|evt.ex.div0|, myErr);
    end;
   end
  onexception |evt.ex.div0| do
   begin
   // invoke your exception handler
   end
end
```

Utility Functions

The current NewtonScript implementation supports a large number of utility global functions, including a test for the existence of a variable or slot:

```
if aVariable exists then ...
if aFrame.aSlot exists then ...
```

The latter can also be expressed using the HasSlot() global function, which exhibits no inheritance

```
if HasSlot(aFrame, aSlot)
```

or the HasVariable() global function, which searches according to the prototype and parent inheritance:

```
if HasVariable(aFrame, aSlot)
```

There are also equivalent calls for getting slots, using GetSlot() and GetVariable(). GetVar(aSlot) begins a slot search in the current frame.

The Newton Toolkit (NTK)

The first version of the Newton Toolkit, available only for the Macintosh, is an integrated development environment which incorporates a project manager, visual interface editor, method browser and editor, byte code compiler, downloader, monitor-debugger, and various utilities. At the time of writing, the NTK contains beta software and alpha documentation.

The NTK uses a Newton connected (or "tethered") to the host Macintosh by a serial or LocalTalk connection. Although some have found the LocalTalk option successful, the serial option seems to be more reliable across a range of Macs (Apple concedes that the LocalTalk option is not yet fully tested). The trickiest part of installation is establishing successful communications with the tethered Newton–you must pay attention to all the documentation in order to avoid frustration. Once established, communications between the NTK and the small debugger-monitor on the Newton (installed by the NTK) are usually good and very reliable. Each time that you open the beta NTK there is a long delay while the global definitions file (a text file called GlobalData, well worth looking through) is read.

Hardware requirements are stringent–a 68030 or 68040 Macintosh with at least 8 Mbyte of memory (and achieving that with virtual memory results in very slow operation), running System 7.1 (or 7.0.1 with TuneUp 1.1.1) in 32-bit memory mode. The NTK likes a Quadra to be truly usable, although it can run on PowerBooks, the SE/30, and Color Classic.

Projects

The NTK uses the same project concept as Symantec's Macintosh languages tools. Opening a project displays a window showing a list of files: layouts containing the view specifications, prototypes which may be loaded from those layouts, print formats, and resources. You can set project preferences to control debug code, optimization (for space or speed), version numbers, names, and so on. There are also preferences for defining the Newton for which the project is configured (currently the only one supported is the MessagePad).

Applications built with the NTK and downloaded to a Newton appear in the Extras drawer and use a default icon. To add a custom icon, you need to create a PICT (32 by 32 pixel) resource containing the black and white icon (assign it a name–critical information–as well as a number), drop it into the project folder, and then add it to the project using the Project/Add file... command. If you create an invert region icon with the same name but the exclamation mark '!' appended, your icon inverts correctly when it is clicked. To use the resource file by you need to open the Project Preferences dialog and indicate that your new PICT should be used as an icon.

Editing Views

Double-clicking on a layout or prototype file opens that file in the NTK's superb visual editor (see Figure 2). The editor displays a virtual Newton screen (which can be resized to allow for other Newton models) and the palette of available view prototypes and classes. To add a new view object to the open window you click on the correct iconic button in the palette and then drag the icon to the editor window. You can also add custom prototypes to the user prototypes button towards the top of the tool palette. Above that there is a pop-up text menu of built-in view classes and prototypes corresponding to the iconic tool buttons.

The view editor window has two modes–standard editing mode displays a point grid and information about the views, preview mode displays the window as it is seen on the Newton. Unfortunately, the latter is not currently truly WYSIWYG. It doesn't display text data particularly accurately, or perform the links that would appear if the application were running. There are also some irritating errors in the positioning of some view objects, leading to incongruencies between design intention and Newton results.

The palette provides alignment tools. There are also positioning controls including dragging to snap to grid with the mouse, and cursor keys to step single pixel increments. You should be careful with copying and pasting views in the beta release, although duplicating view objects appears robust and reliable. To access a view's methods and slots, you must name the view and access it from the (named) base view.

Slots and Methods

To open the slot editor/browser, shown in Figure 3 on the next page, option-double-click on a view. Although the slot editor's interface has some rough edges, it should be familiar if you've used MacApp's Mouser/MacBrowse or similar tools. Double-clicking on a slot name brings its contents into the lower area of the browser, ready for editing. There are standard templates to lay out the contents of simple data types such as numbers and Booleans, as well as common compound types such as viewFlags (shown). All the views within the browser window support live scrolling of their contents as you drag the scroller control.

You use the same browser to edit methods, represented as simple text code embedded as a slot. Currently, formatting for such NewtonScript code is primitive, with no automatic tabbing (for instance). New slots are easy to add–those specific to the view type are shown in a pop-up menu in the browser, along with the main view methods and data slots. You can name other slots and assign each of them one of the built-in, type-specific slot editors.

Compiling and Testing

NewtonScript is an interpreted byte-code language on the Newton. The NTK has to compile your code to byte codes, build the complete package for downloading to the Newton, and provide a downloader and debugger. The default compile command saves source files first. This

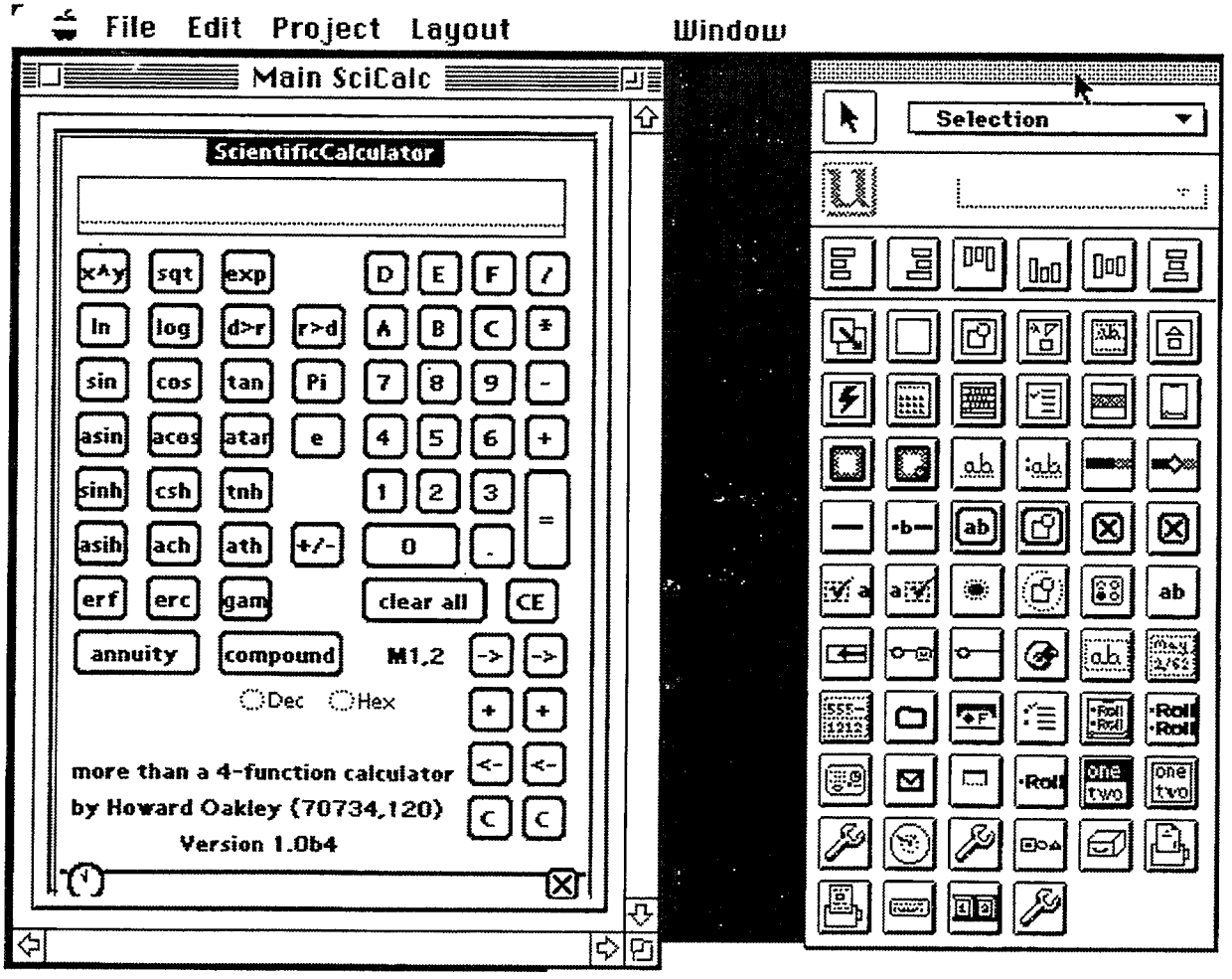


Figure 2 - The NTK layout editor in preview mode.

step plus compiling can take several minutes as a project file grows. The NTK automatically creates (or updates) a Newton package file within the project folder, which is in turn found automatically by the downloader.

The easiest way to download an application for testing is to open a debugger/inspector/monitor window; connecting such an inspector requires minimal work to support downloads. If an inspector is not connected, each download requires synchronized commands on the host Mac and the tethered Newton. Connecting an inspector requires just one set of synchronized commands, after which you can carry out an unlimited number of downloads simply by using the Command-2 key combination on the Mac. Since the default is to remove the old application when downloading a new version, testing can be remarkably slick.

Debugging

A connected inspector can display the results of Print() commands embedded in an application, can accept debugging commands such as calling up a stack crawl or invoking a trace, and can evaluate NewtonScript commands directly. It is thus possible to carry out highly interactive debugging sessions in which errors are located and fixed before fixing the source code.

Novices usually start by using two of these techniques. Print() enables you to output any string, including variable values or code place markers, to the inspector. Using this technique you can trace progress through a method by inserting Print() calls at appropriate points to locate an error. Turning function tracing on, using

```
trace := 'functions
```

prints to the inspector an ongoing text account of all functions as they are called. This method is very slow. Also, if you try to turn it off using

```
trace := nil
```

you may have to wait until there is a pause in the text stream.

The ability to enter any NewtonScript command is invaluable, and enables you to sniff around inside the Newton itself. For example, you can look at the root object using

```
getRoot()
```

and get a listing of global functions using

```
printdepth := 0;    self.functions
```

The only major caution when using the inspector is that you submit a line to the interpreter/debugger by pressing the Enter key rather than the Return key. Failing to

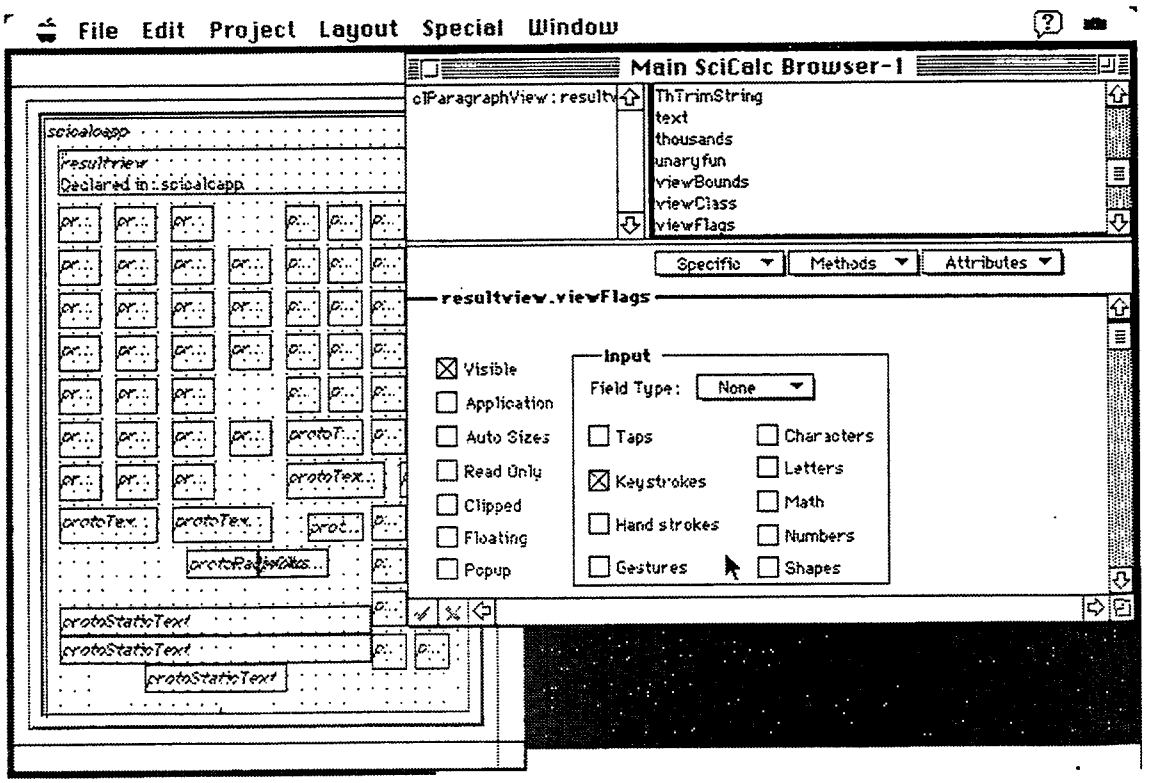


Figure 3 - the NTK editor/browser showing a viewFlags slot being edited.

do so has confused and delayed many developers. If you want to submit multiple lines at once, you should select all of them before pressing the Enter key.

Providing Application Data

One task which has provided troublesome, largely because of the lack of documentation, is moving data–text, sound, pictures, and so on–from the host Mac to a Newton application *[see Downloading Data to Your newton on page 25]* . Resources can be read into NTK projects by including their resource file (typically, from ResEdit) as one of the project files, then adding code similar to the following, to the Project Data:

```
rf := OpenResFileX(":myProject:myData.rsrc");
   // note the proper path
gPictureA := GetResource("PICT", 1000,
             'picture);
   // for PICT resource 1000
CloseResFileX(rf);
```

(Note that you do not need to use OpenResFileX() and CloseResFileX() for PICT resources, but you must use them for sounds and other resource types.) In the main application view, an evaluation slot named myPict, with a value of gPictureA, then loads the data and makes it accessible.

An alternative, for text and numerical data, for example, is to declare them in the Project Data as simple equates:

```
myText := "A sample piece of text";
```

and then to add an evaluation slot to the main application view with a value of myText. There are many variations to this approach, but individually declared items (excluding resources that are read in) in Project Data cannot exceed about 7 Kbytes in size.

Conclusion

In adopting the Newton architecture, Apple is addressing the fundamental design differences between desktop computer systems and palmtop consumer devices. Users do not have to wrestle with filing systems which are alien to their experience, and the memory (and thus power) requirements of applications are minimized by optimizing in situ code reuse.

Most comparable pen-computing systems are fairly conventional in their System fundamentals, their choice of programming languages, and their development systems. Given the wide gulf between previous computer systems and PDAs, this is surprising. It encourages unpolished attempts to port applications from computers to PDAs, and it also suggests that few manufacturers consider the requirements of PDAs to be any different from those of previous desktop and mobile computers. This is an approach which lacks vision and which bodes ill for their future.

In contrast, Apple has rethought hardware and system software design from the ground up, and as a consequence has delivered something different in every respect from a Macintosh. To have achieved this and developed a new–and highly effective–language along the way is a remarkable feat. Although it is too early to assess whether Newtons will realize their promise, they and their development tools are a joy to work with. I heartily recommend them.

Further Reading

For a while at least, the only in-depth reference to the Newton ROM and NewtonScript is the Newton Toolkit manual. A number of additional books are in preparation.

The ARM610 processor is described in an article in *Byte* by Dick Pountain (December 1991, International Section only, pp 84IS-49 to 60), and is documented in standard data sheets (the *GEC Plessey Semiconductors ARM610 Data Sheet*, 1993, publication no. DS3554).

SELF is described in detail in a number of papers and reports by Unger, Smith, Chambers and others, which are available by ftp from self.stanford.edu, where executables for Sun 3 and 4 systems are also provided.

Acknowledgements

I am greatly indebted to the many members of the CompuServe Newton forum for discussions and support, particularly to Nigel Ballard and those Apple employees who gave up their own time to support forum members. Kent Sandvik of PIE Developer Tech Support very kindly read and corrected this article. π

Howard Oakley is a full-time Mac and Newton developer, and a part-time medical researcher. His development company has a range of specialist CAD/CAM applications in use primarily in the sailmaking industry. He wrote one of the first Newton freeware utilities, and has several commercial applications in early development.

In adopting the Newton architecture, Apple is addressing the fundamental design differences between desktop computer systems and palmtop consumer devices.

Howard Oakley

Getting Started Q & A

How do I become a Newton Developer?
First, buy the Newton Toolkit (NTK) from APDA, which costs $795 + taxes + shipping. The version currently available requires a Macintosh, but a Windows version is under development. Then, register your NTK. On registration, you should get a mailing regarding developer support. If you live in a country which has non-US developer support for other Apple products, then you should also contact your local Apple Developer Services contact.

What's this about royalties on Newton products?
Newton technology, including that in the NTK, is licensed from Apple. If you want to license hardware or firmware products for Newton, such as PCMCIA cards, then you should contact Apple's PIE Division for special information relating to those. If you want to use the NTK for commercial development, which includes shareware, then you will need to complete a licensing agreement. One provision of this is that 1% of net commercial product revenues will be paid as a royalty to Apple. The situation regarding shareware is not so clear, and details are still being worked out. Of course, freeware remains free. In-house developers can purchase internal distribution licences appropriate to corporate needs.

What about support?
Details of Apple's support programs are being finalized now. They will be offering a three tier system. The most basic level (Pioneer) is included with the NTK. If you require one on one support, then there is a Partner program which costs around $2850, but brings with it commensurate benefits. Contact the Developer Support Centre (USA) at 408-974-4897 if you want further details. A corporate support plan caters to the special needs of those working on in-house projects. There is also ample informal support available through AppleLink, America Online, CompuServe (GO NEWTON), the Boston Computer Society, BMUG, and other user groups. Apple does have formal and informal representation on several of those.

I can't do what I want with NTK. Is there an alternative?
Are you really sure that you can't with the NTK? Is it a feature–like the BookMaker interactive book support–which is not yet available? Are you working on communications, which currently need an ARMistice NuBus card, but which will be supported by the NTK in the final release? It may be that your product requires access to other Newton technology, in which case, you need to talk to Apple PIE, and make your case to them on AppleLink NEWTON.DEVS, or Internet NEWTONDEVS@newton.apple.com. Are you coding inefficiently? Perhaps you really do need access to C++, but you need to make your case to Apple.

What about vertical markets?
Apple has confirmed its interest in vertical market software, but appreciates that you know (or should know!) your market. It is therefore, for the time being, concentrating on encouraging horizontal and content-based products, and hardware. You are therefore welcome to join one of the developer programs, but Apple does not for the moment have the same marketing interest as they do in horizontal applications.

Do I need to get every Newton PDA, including Sharp's?
Apple's hope is that you should not, but I'll bet that you will find it hard to resist! New machines which are significantly different from the MessagePad will be supported with new drop-ins for the NTK, designed for different screen sizes and other features. There are no differences in software between the current Apple and Sharp Newtons. Software developed for one should work just fine for the other.

How do I get a screen shot of my application?
You currently need an ARMistice card, I am afraid. The problem is not just getting the shot (although you might find Command-Shift-3 a bit tricky!) but getting it back to a host Mac. This is being worked on. There should be an answer fairly soon.

What other tools are there?
None are generally available, apart from the NTK. Macintosh development tools are a mess, and Apple seems determined to get the Newton right with a proper integrated environment this time, which should take care of the great majority of developers' needs. Those who do feel they have a case for using C++, or who need to write drivers (such as for PCMCIA cards), should first familiarize themselves with the Newton through the NTK, and then contact Apple PIE. However, Apple does intend to produce a simpler and cheaper 'user' programming environment, once the NTK is fully featured and mature.

Where can I get code examples and books?
There are a number of books coming out now. One of the first inside accounts is *Newton's Law* from Random House, and there are at least two in preparation on developing for the Newton. Example code is provided with the NTK, and is available on the many online services. But best of all, your subscription to *PIE Developers* will keep you up to date with the very latest technical information.

What's all this about Llamas?
Watch the film "Monty Python and the Holy Grail". π

What Skills Do I Need To Program A Newton?

by Jim Lanford
Micro Dynamics Ltd
72435.442@compuserve.com

The Newton is great fun to program. Doug Olson from Adobe, author of "KidsCard," writes in his About Screen that KidsCard "was quick, easy and a blast to do". But before you spend $800 for the Newton Tool Kit (NTK), and another $700 for a Newton (if you don't already have one), you might want to know if you have what it takes to write great Newton software, what skills you'll need, and how you might go about getting these skills if you don't have already them.

I started by looking at what skills Apple thinks are necessary. All I could find is the somewhat terse prerequisites that Apple's Developer University gives for its Newton Training class: "Familiarity with the Macintosh System and one year experience with Object-Oriented Programming" (OOP). That's a start, but it's not enough. Since the Newton started shipping in August, programmers have been asking me if they have the necessary background to start programming the Newton.

What's the ideal background for a Newton programmer?
If you are multi-lingual in the computer sense, love to learn new environments and languages, intuitively understand objects, have experience with browsers, know several Macintosh drawing tools well, thrive on "bleeding edge" documentation and tools, and love playing with the Newton, then you clearly have what it takes to write great Newton software. If fact, you may as well skip to the next article or start writing your first program.

If you don't have all of these skills and qualities, don't feel badly. Very few people do. Let's look at what skills are important and how to learn what you need to get going.

Do I have to know more that one computer language?
Being multi-lingual is an easy way to determine that you can quickly learn another computer language. I know programmers who love one language and are miserable using another.

You program the Newton using NewtonScript. If you know several other languages, and at least one of them is object-oriented, learning NewtonScript should not be too hard. The language section of the NTK manual is only 36 pages and the reference card is only two pages. NewtonScript is closer to Pascal than C.

NewtonScript seems to be a descendant of a language developed at Stanford called SELF, according to Howard Oakley (author of Memory Massager and other Newton shareware). Some Newton programmers have found that reading SELF documentation has helped them understand NewtonScript. If you have internet access, you can ftp the SELF server at "self.stanford.edu".

Why is experience with OOP necessary?
Developing for the Newton means living in an object-oriented environment. If you understand object-oriented concepts such as classes, inheritance and encapsulation, you will have an easier time if you don't have to learn these profound concepts while learning a new language. Most of your time will be spent learning all the new Newton classes that come with NTK. You will also learn some potentially new concepts: frames, slots, soups, and stores.

Objects are new to me, but I know several procedural languages. Do I need extra training or is the NTK package enough?
In its current state, the NTK package does not have a basic object tutorial. This would be a great opportunity for a third party book or perhaps an ongoing column for this journal.

One of the fastest ways to learn objects is to go through the excellent tutorial that comes with the Digitalk Smalltalk/V package. You will learn OOP and how to use a browser, which will help you a great deal in understanding and using NTK. A more formalized and in-depth way to learn Smalltalk is to take ParcPlace's one-week class on their Objectworks Smalltalk-80 product. I know several C developers who got right into C++ after taking this class.

A few Newton programmers have recommended another way to learn objects, which is through APDA's "Introduction To Object-Oriented Programming" self study class. This course includes a video tape and CD and costs $295 (APDA No. R0001LL/B)

I heard some developers say that learning MacApp is a great way to learn objects and browsers. I think that this would be too time consuming unless your goal is to program the Mac. But if your goal is to learn objects so that you can program the Newton, the Digitalk Smalltalk/V tutorial is a much faster way to learn.

I'm not a graphic artist, so why do I need to know drawing tools?

When using NTK, one of the first things that you do to start writing a new application is to select the proto template icon from the layout window palette and draw the application base view in the layout window. If you know drawing tools on the Mac, you don't even think about doing this.

Knowing how to generate, acquire, crop, dither, scale, and convert graphics on the Macintosh makes adding graphics, icons and other finishing touches to your Newton project much easier. It helps if you have several favorite graphic tools that you already know how to use. My favorite graphic programs are Canvas, Photoshop, DeskDraw and DeskPaint.

You also have to use Apple's ResEdit utility to put your graphics into one file that you add to your NTK project. Hint: Make sure to name the PICT resources or the NTK browser won't display them.

You mentioned a browser. What is it?

In an object-oriented environment, a browser can be thought of as an intelligent object-aware editor. The NTK browser allows you navigate your templates and slots easily. After you draw the application base view as mentioned above, you use the browser to fill out or add slots to your prototypes. Writing NewtonScript involves filling out the appropriate template in the browser. For example, you'll find the viewBounds.Rect values filled with the values from your drawing in the layout window. In other slots, you simply change the default text.

If you are used to typing everything yourself, you'll really like using a browser.

Loving the Newton...

Programmers who love their target environment usually produce much better programs than those who don't. If you love the Newton, you will work around its flaws and attempt to produce software for it that emphasizes its strengths.

If you don't think the Newton is fun, and you agree with Gary Trudeau's "egg freckles" type review of the Newton, don't bother to try to learn it. If you think that C is the only language, and command line interfaces are mandated from on high, but you have to program the Newton because your boss told you to... forget it Any Newton software you write will be mediocre at best.

If you think that the Newton is the start of a new generation of digital tools, and have a tolerance for some frustration, the reward will be a lot of fun building Newton applications. π

Jim Lanford is the Founder and Chief Technical Officer of Micro Dynamics, Ltd, a member of the INC. 500 for the last two years. He built his first micro in 1975, has been a Macintosh developer since 1984, and has been programming the Newton since August.

Which Mac Do I Buy To Develop For The Newton?

Jim Lanford

If you own a PC and don't want to wait for the MS Windows version of the Newton Tool Kit (NTK) you need a Mac. Since Apple's current product line is very confusing., here are a few ideas about which models to buy.

NTK requires that your Macintosh has a 68030 or 68040 and four Meg of available RAM. In other words you'll need 8 Meg of RAM and at least a 40 Meg hard disk.

If you decide to buy a new Mac and you want to spend as little as possible, then look at the Color Classic. The street price for a Color Classic 4/80 is currently $1,050. Add four Meg of RAM for between $100 and $200 (RAM prices vary daily). That is all you need.

Another option that is often overlooked is the Centris 610. The street price is $1,280 for a 4/80 610. There are rumors that the Centris 610 will be renamed to Quadra 610. You'll have to add a keyboard for $100 and four Meg of RAM. If you have a VGA monitor, you just need a $15-$20 Mac-to-VGA adaptor cable. I know one developer running a $100 "paper white" VGA monitor on a Centris 610. You can have a much faster Mac than the Color Classic for only $350-$450 more.

Another alternative to consider is used Macs. Unfortunately used Macs do not drop in price as much as some people think. That is because older models have more Nubus slots than the newer low-end Macs. If you do find a good deal on a used Mac, you should consider purchasing AppleCare. AppleCare is Apple's warranty program that covers all parts and labor on Apple equipment. As a condition of sale, take the used equipment to an Apple Dealer and have the dealers's technician certify that the equipment is in working order so that you can purchase AppleCare. If the Mac you buy starts falling apart, you won't have to pay for repairs.

You might also try asking your Macintosh fanatic friends. They may have an extra Mac that they are planning to give a niece or nephew to use at college next June. You could borrow it, hoping that the Windows version of NTK is shipping by then. π

MacsHex-A Second Newton Application

Application Cookbook

Mac Foster
Object Factory Incorporated
OBJFACTORY@applelink.apple.com

Everyone's first Newton application is the Hello World tutorial (if you skipped that step, come back when you're finished with this). With one project under your belt, you're ready to tackle something more original. For my second project, I decided not to dive right into GalacticCommWhiz 2000, but to try something simple and understandable that would do something useful, albeit totally retro. It has always irked me not to have a hex calculator handy while debugging on the Macintosh, and what could be handier than Newton? So I decided to make that my second project.

As I'd hoped, I learned in the process. First, I needed to bring the basics of NewtonScript, views, templates, parents, and protos into much sharper focus to actually write code. The manuals are sketchy when it comes to basic examples. The result is easy to see in early code: lots of superfluous view naming, SELF reference and GetRoot() calling. As better documentation arrives, this confusion should wane. In the meantime, I have included some notes about conceptual hurdles I had to jump.

Second, I had to adjust to the machinery of the Newton. To overcome the limitations of 30-bit Newton integers, the project had to use real numbers and do its own conversion to strings for display purposes. This involved using the built-in SANE library, which was hardly documented except for a list of functions. I had only used SANE for a few things on the Macintosh, and, rather than run out and buy an Apple Numerics manual, I had to operate in try-it-and-see-what-happens mode on the Newton. Fortunately, this was not a problem. The Inspector is great for this kind of exploration and the SANE functions are intuitively named. Even though I had some background with SANE on the Macintosh, a developer with a Windows background shouldn't have any more trouble. SANE on the Newton was actually easier than on the Macintosh because I didn't have to worry about data representation. In the body of the article, I have described the issues that did arise.

This was enough to "finish" the project. In the month since, Apple has come to grips with some issues regarding use of the global name space inside the Newton. In the version described in this article, I have gone back and applied the latest naming conventions, so you can see how it is done.

Clearing the First Hurdles

A few Newton basics didn't seem to leap right off the pages of the manuals when I read them. It wasn't that I couldn't write code without knowing some of these things, just that I had no idea what I was doing. In hindsight, most missing information revolved around naming. Hopefully, this section will spare you some of that early confusion.

Sending A Message

Let's start with the simplest thing I can imagine, sending a message from one method to another in the same frame. I looked in vain for an example in the manual. The syntax is:

```
:MethodName(arguments)
```

The colon is required to distinguish a method name from the name of a global function. If you do not specify a frame, as above, NewtonScript starts looking for the method as a slot in the current frame. This syntax tells NewtonScript to use the default search—current view, the _proto chain of the view, the immediate _parent, and so on, applied recursively.

If, instead, you use the name SELF, like:

```
self:MethodName(arguments)
```

the parent chain is not searched. You can get quite confused if you think calls with and without SELF are the same.

There are lots of examples of sending messages to named views in the manuals. The syntax is (as you know):

```
ViewName:MethodName(arguments)
```

This works like a SELF search, and seems obvious. Not so obvious, at least to me, was the difference between naming a view and declaring a view, which makes the view name usable in a method.

Well I Declare

One of the more interesting things the NTK and Newton environment conspire to do is declare view names. There are a couple of ways to do this. Say you have a parent view named P and it has a child view named C. Open up the Template Info dialog for view C and, in the Allow Access From popup, select view P. This declares C in P

When you declare view C in view P, you tell the Newton runtime to create a slot in view P named C that points to view C. Got it? Now, any time a method in view P wants to refer to a slot in view C or send a message to a view C method, it can just say,

```
C.slotName
C:methodName()
```

This works because the slot named C is located via the _parent chain and evaluated to yield a pointer to view C; the pointer is used to start the search for the slot in C.

Since unqualified slot references are resolved by looking up the _parent links as well as the _proto links, this also means that any child view of P can refer to C with the same ease that P can. So the simple trick that makes a view universally accessible is to Allow Access From the application view. The slot with the view's name then goes into the application view, where every view can find it by looking up its _parent link.

In MacsHex, every button sends messages to the displayView, so it is declared in the application view. No other views are declared.

Declarations Come and Go

The subtle part of declaring views is that the views involved don't exist, at least not necessarily and not all the time. The Newton runtime creates the view pointer slots, if needed, and initializes them with view pointers when declared views are opened. When declared views are closed, the runtime sets the value of the view pointer slots to NIL. You can easily see this in the inspector if you examine the contents of an application view before it is first opened after downloading or reset, after opening and after closing.

The Global View

Only the Newton can make a view. Although the manual suggests otherwise, you don't create views with the NTK, you create templates. When your application is downloaded to the Newton, the application view is immediately created in memory. Its _proto link points to the application view template you defined in the NTK; the template contains a stepChildren array which points to the templates for all the children of the application view. The child views are not created until the user opens the application view.

The application view sticks around in memory through opens and closes until the user resets, whereupon it is rebuilt from scratch. This means it is not a good idea to stuff a lot of runtime data into slots in the application view. These slots do not go away when the application is closed, and their contents hang around occupying memory.

After the application view is created, a slot is added to the root view containing a pointer to it. If the application view were "declared" like other views, you would expect the slot to be named with the application view name. But names inside an application are local to the application. Instead, the slot is named with the application symbol you define in the Project Settings dialog. *[For more details about declaring views, see "Tales From the View System" on page 32.*

Names and Signatures

The application symbol must be globally unique in the root view. If a different application gets downloaded that uses the same symbol, bad things happen. (At least one developer reported that his Newton went completely out to lunch, generating error alerts whenever he tried to do anything. Not the sort of behavior you want associated with your product.)

There was no way to ensure unique global names when the Newton was first released, but in early October Apple jumped into the breach with a registered signature strategy (see the sidebar, Newton Developer Registry Guidelines, on the next page,). In a nutshell, you apply to Apple to register a name called a signature. You get one signature per company, not one per program. You use the signature to construct other unique, internal names. There is a good reason for making your signature a recognizable version of your company's name (see the second bullet point below).

In the screen shot examples in this article, we use the dummy signature "Sig". The actual MacsHex uses Object Factory's signature. You should use your own signature for your projects. Registration is quick, easy and painless; we got our response from Apple in one day. Here are some names you must make unique using your signature:

- The application symbol is used as a slot name in the root view. For example, |MacsHex:Sig|. The application symbol is set in the Symbol field of the Project Settings dialog (see Figure 1 - Project Settings, on page 16).
- The package name is used to identify your application package internally. It's not just an internal name; it's visible to the user in the Remove Software menu. Set it in the Package field of the Project

Continued on Page 16

Newton Developer Registry Guidelines

To avoid name conflicts among Newton applications we are having developers register a single signature. You do not need to register individual applications. Once you have a signature, the guidelines below explain how to ensure the various global symbols and string your application uses will be unique.

SIGNATURE
Example: PIEDTS
Signatures are an arbitrary sequence of approximately 4 to 10 characters (colons and vertical bars are not allowed, case is not significant).

APPLICATION NAME
Example: "Llama"
This is the string displayed under your application's icon in the Extras drawer. This symbol does not have to be unique. So it would be possible for there to be two applications named "Chess" on the market. Presumably, their icons would distinguish them if they were both installed.

APPLICATION SYMBOL or appSymbol
Example: '|Llama:PIEDTS|
The appSymbol is formed by concatentating the application name, a colon, and the registered signature. This symbol is not normally visible to the end user. It is used to uniquely identify an application to the root view. Note: since appSymbols contain a colon, they have to be enclosed in vertical bars where they appear explicitly in NewtonScript code.

PACKAGE NAME
Example: "Llama:PIEDTS"
The package name is simply a string version of the appSymbol. The package name, is visible to the user in the pop up menu for removing software. Package names are limited to 26 characters, so this places a practical limit on the combined length of application names and signatures.

PRE-DEFINING CONSTANTS
We suggest that you define constants for your appSymboland package name in your "Project Data" file.

```
Example: constant kAppSymbol:= '|Llama:PIEDTS|;
         constant kPackageName := "Llama:PIEDTS";
```

Use these constants throughout your code instead of hard coding in values. This will make it easier to maintain your code. All you have to do is keep your constant definitions consistent with the values in NTK's Settings... dialog. A future version of NTK may automatically define these constants for you.

For the remaining examples we will assume the above constant definitions are in effect.

SOUP NAMES
Examples: "Llama:PIEDTS" kPackageName
"soup1:Llama:PIEDTS" soup2" & kPackageName
The soups your application creates need to have unique names. You can use your package name directly or along with a colon as a suffix to a descriptive soup name. Soup names are not normally visible to the user. They are displayed to users by Newton Connection.

SYSTEM SOUP TAG
Example: `{tag: kPackageName, firstName: "Reg"}`
Applications are expected to store their preferences data in the system soup. Each application is supposed to create a single soup entry with a unique tag slot and other application specific slots. Use your package name as your tag. This tag is not normally visible to the end user. However, if utility software is written to "clean up" the System soup, this tag may be used to prompt the user asking whether or not to delete an entry.

ADDITIONAL SOUP FIELDS
Example: `entry.(kAppSymbol):={<AppSpecificData>};`
Applications are encouraged to utilize existing soups, like the Names soup. If applications needs to store their own data in entries of an existing soup, they should create a single slot using their appSymbol as the slot name.

ROUTING FORMATS
Examples:

```
GetRoot().('|fmt1:Llama:PIEDTS|):={<Fmt Frame>};
GetRoot().(Intern("fmt2:"&kPackageName):={<FmtFrame>};
```

Applications need to store their formatting frames in the root view. Each formatting frame should be given a name and stored in the root view in a slot whose name is the format's name concatenated with a colon and the appSymbol.

GLOBAL VARIABLES
Example:

```
GetGlobals().(kAppSymbol):=<AppSpecificData>;
```

Although the use of system wide global variables is strongly discouraged, it's sometimes necessary to use them. For this purpose, your application should create a single repository for global data by adding a slot to the global's frame using its appSymbol as the slot name.

HOW TO REGISTER
To register your signature, please send the following information to PIESYSOP on AppleLink or PIESYSOP, MS/305-2A, 20525 Mariani Ave, Cupertino, CA 95014

Company Name:
Mailing Address:
Contact Person:
Phone:
Email Address:
Desired Signature 1st choice:
Desired Signature 2nd choice:

MacsHex - Continued from Page 14

Settings dialog. Apple suggests that developers make the package name a string version of the application symbol, such as "MacsHex:Sig". This package name, by the way, has nothing to do with the name of the package file created by the NTK.

- Soups, preferences, and application globals need unique names, as well. MacsHex does not use any of these. See the sidebar for details.

DEBUG Names

You might think (wish?) that there is nothing else to say about internal names. But here's a tip we didn't find in the manuals. When you name a view, the NTK automatically makes a slot named Debug in the view that contains the view name as a string. In the Inspector, the Debug() function called like:

```
v := Debug("viewname")
```

returns a pointer to the view. Very handy.

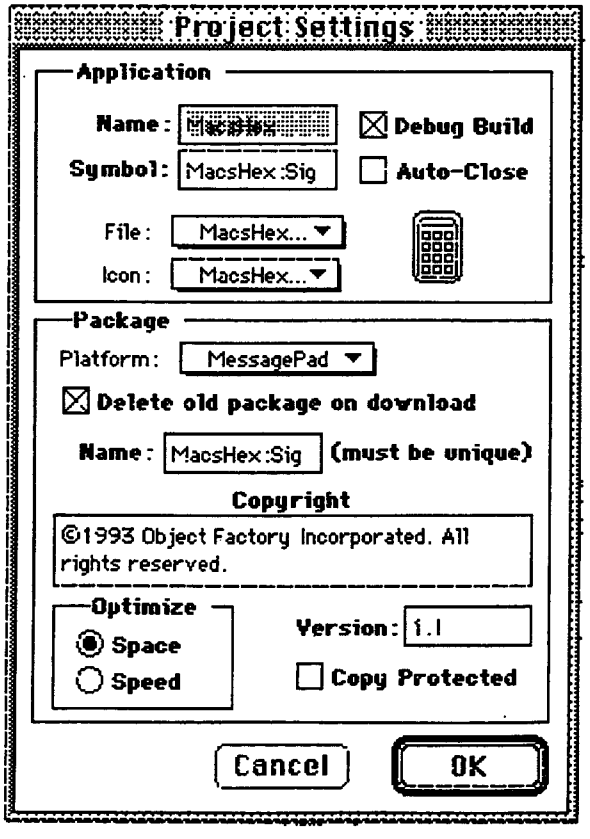


Figure 1-Project Settings.

Names the Customer Sees

The two most important names to your application's users, of course, are the trademark names—the application name and its icon in the Extras drawer. You create an application name in the NTK by typing it into the Name field in the Project Settings dialog (see Figure 1-Project Settings.)

The developer must provide an icon for display above the application name in the Extras drawer. The NTK actually doesn't use icons at all, but PICTs. Your application icon must be in the form of one or two named PICT resources, the icon itself, and an optional mask. The simplest way to create these PICTs in ResEdit is to draw the icons as ICON resources, then Cut each icon, open the PICT window, and Paste the icon as a PICT. The PICT resources must be named; the mask name must be the same as the icon name with a '!' at the end (such as "Name" and "Name!").

The next step is to add the resource file containing the icon PICTs to your NTK project. Open the Project Settings dialog and, in the File field, select the name of the resource file containing your icons. Then, select the name of the PICT resource in the Icon field.

MacsHex

Overall Design

I named MacsHex's application view HexCalcApp. This view holds all other views in the calculator. These include a clParagraphView to display results and a lot of _protoTextButtons that act as keys for the calculator. All the button views look alike, so they are all named (see Figure 2 - MacsHex Layout.)

There is also a linked subview in MacsHex. This is linked to a layout containing a _protoFloatNGo. I used the linked layout as an alert, which opens if the user attempts an operation that cannot be done (such as divide by zero). The layout comes directly from the NTK's tutorial project and is not shown here.

Button Views

The text buttons all call methods in the displayView and pass their value to the method. Number keys such as the 'A' key call pressNum in their buttonClickScript.

```
A.buttonClickScript
func()
begin
  displayView:pressNum(10.0)
end
```

The operator keys call pressOp and pass a string representing their operation.

```
=.buttonClickScript
func()
begin
  displayView:pressOp("=")
end
```

There are also two buttons labeled clear and all clear whose buttonClickScripts send corresponding messages to the displayView.

DisplayView

I used a clParagraphView named displayView as a display for the calculator. Because this is the only view which must be sent messages I put all the methods and value slots in the displayView frame. The slots curNum and StoreNum hold the most current number entry and the previous total respectively. There is also a slot named curOp which holds the most recent operator pushed (stored as text). The Boolean slot opLast remembers whether or not an operator was the last key pressed.

There are also slots holding methods to be called by other views. The method PressNum is called when a pen down occurs in a number key. It changes the value of curNum and displays it in displayView. Because this is a 32 bit calculator, the Remainder function is used to keep the number smaller than 232.

```
PressNum
func(theDigit)
begin
  local  hexNum;
  curNum := Remainder((curNum * 16) + theDigit,
                      4294967296.);
  hexNum := :HexString(curNum);
  SetValue(self, 'text, hexNum);
  opLast := NIL;
end
```

PressOp is called when an operator key is pressed. It first checks to see if an operator was the last key pressed. If not, the pending operation is performed using a function that tests for equal strings (if = were used the string pointers would be compared instead of the string values). The Remainder function is used, again, to keep the result in the proper range.

```
PressOp
func(key)
begin
  local  hexNum;
  if not opLast then
  begin
    if StrEqual(curOp, "+") then
      storeNum := storeNum + curNum
    else if StrEqual(curOp, "-") then
      storeNum := storeNum - curNum
    else if StrEqual(curOp, "=") then
      storeNum := curNum
    else if StrEqual(curOp, "*") then
      storeNum := storeNum * curNum;
    else if StrEqual(curOp, "/") then
    begin
     //Check for divide by 0 and open alert
     if curNum = 0.0 then
      begin
        linkedhex:open();
        self:initsFunc();
      end;
     else storeNum := Trunc(storeNum / curNum);
    end;
    storeNum := Remainder(storeNum, 4294967296.);
    hexNum := :HexString(storeNum);
    SetValue(self, 'text, hexNum);
    curNum := 0;
    opLast := TRUE
  end;
  curOp := key
end
```

Continued on Page 22

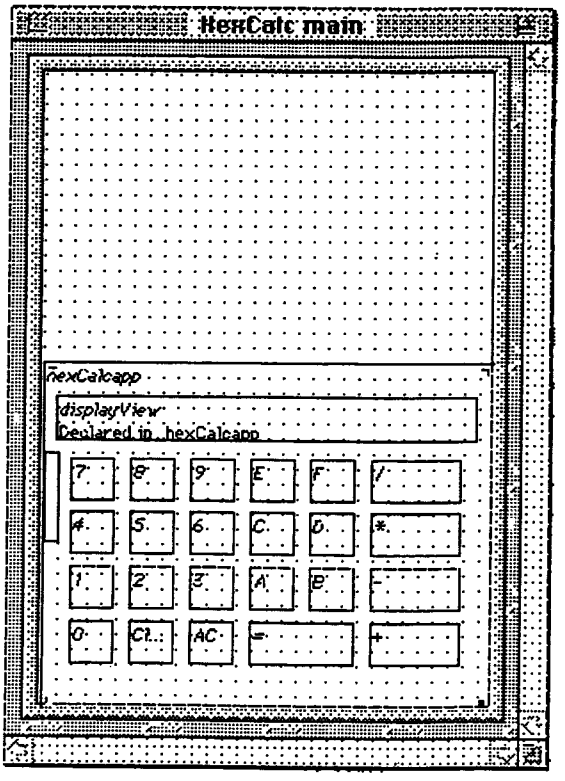


Figure 2 - MacsHex Layout.

Howard Oakley

SciCalc– A Scientific Calculator

Application Cookbook

Although the Apple MessagePad (and Sharp ExpertPad) come with a basic four-function calculator, there was early demand for a scientific calculator. This promised to be a good initial project, and my first explorations with the Newton Toolkit (NTK) suggested that the ability to compile and execute Newtonscript on the fly could be very useful.

Design Model

My intention was to have a single view containing the numeric display, which had as method slots all the functions needed to run the calculator engine. This view would hold the current text in the display, its numeric value, and the previous numeric value, as well as other slots such as the contents of two memories. The calculator buttons would then hold simple methods to call routines in the result view as they needed to perform their functions.

Within the result view methods, evaluation would consist of building a Newtonscript string to perform the calculation, then calling compile() to add the method as a temporary slot, which when called would evaluate it:

```
// compile the code into a temporary slot
self.temp := compile("2+3");
a := self:temp(); // then evaluate it
```

This should result in a very compact implementation, with the modest compilation overhead going unnoticed given user interaction times.

Classes and Views

The whole application was set in a protoApp application view, given that it was going to be too large (in terms of display area) to be useful as a floater. I named the protoApp view 'scicalcapp' to make certain contained views were accessible from other views.

The result view was made from the template clParagraphView, with the viewFlags set as follows: Visible TRUE, Input Keystrokes TRUE, and field type None, all other flags NIL. This view was named 'resultview' and declared as accessible from scicalcapp to make its method slots callable from all other views.

The calculator buttons were each made as protoTextButtons, although I had originally hoped to use a protoKeyboard or protoKeypad prototype to make this simpler. In addition to the immaturity of protoKeyboard and protoKeypad in this beta release of the NTK, I needed to be able to size and place keys individually. This was possible only by making each button a protoTextButton.

Handling Button Clicks

One of a group of methods is called for each button you tap. Each sets any appropriate slots in resultview and then calls the appropriate method. Tapping one of the numeric buttons simply passes the appropriate digit string. For instance, the buttonClickScript for the 7 button is:

```
func()
begin
  resultview:clearzero("7");
end
```

A lot of the scientific functions are unary, and are demonstrated by the buttonClickScript for Cosh:

```
func()
begin
  resultview:unaryfun("Cosh");
end
```

Binary functions like the '+' button have the following buttonClickScript:

```
func()
begin
  resultview:fourfun("+");
  SetValue(resultview, 'isFour, TRUE);
end
```

Pressing the enter key calls the following buttonClickScript:

```
func()
begin
  resultview:evaluator();  // call the evaluator
                           // then reset the slots
  SetValue(resultview, 'operator, "0");
  SetValue(resultview, 'isFour, TRUE);
  SetValue(resultview, 'currentNum, 0);
  SetValue(resultview, 'lastNum, 0);
end
```

Inserting constant buttons for values such as pi is simple:

```
func()
begin
  SetValue(resultview, 'text, "2.718281828");
  SetValue(resultview, 'isNewEntry, TRUE);
end
```

The resultview Engine

The resultView engine has the following notable slots:

```
resultview.text
// the current text displayed in the view
resultview.currentNum
// the current numeric value (a float)
resultview.lastNum
// the last numeric value (a float)
resultview.operator
// a string containing the current operator
resultview.isFour
// flag, TRUE if current operation is binary
resultview.isNewEntry
// flag,TRUE if display is a complete new entry
resultview.mem1      // float memory 1
resultview.mem2      // float memory 2
resultview.thousands
// a string containing the thousands separator
```

If you enter a digit into the display, the following method is called with the digit (as a single-character string) passed as the parameter:

```
resultview.clearzero:
func(theDigit)
begin
  local a, b, c, d;
  if theDigit = "." then
  // first, handle entry of decimal points
  begin
   if isNewEntry then
    // that means this is a new number entry
    begin
      a := "0.";
      SetValue(self, 'isNewEntry, NIL);
    end
   else
    begin
     if (StrPos(a, ".", 0) = NIL) then
      a := self.text & ".";
    end;
  end
  else if isNewEntry then
    // handle other chars being entered anew
    begin
      a := theDigit;
      SetValue(self, 'isNewEntry, NIL);
    end
  else a := self.text & theDigit;
  // otherwise just concat the new digit
  if a then
    // this routine keeps leading minus,
    // points, etc., in order
    begin
      if (a <> "") then
      begin
        b := StrLen(a);
        if (a[0] = $-) and (b = 3) then
        begin
          if (a[1] = $0) and (a[2] <> $.) then
          begin
            c := SubStr(a, 2, (b - 2));
            a := "-" & c;
          end;
        end
        else if (a[0] = $0) and (b = 2) then
        begin
          if a[1] <> $. then
          begin
            a := SubStr(a, 1, (b - 1));
          end;
        end;
        SetValue(self, 'text, a);
      end
      else SetValue(self, 'text, "0");
      // if we cannot make sense,
      // set display to 0
    end;
  end
```

When a unary function is called, the following code is executed:

```
resultview.unaryfun:
func(theFun)
begin
  local a, b, c, err;
  err := -48450; // default error code
  try // this wraps our exception handler ploy
   begin
   c := self.text;
   c := :ThTrimString(c);
   // first, the string acceptable (see below)
   self.temp := compile(theFun & "(" & c & ")");
```

```
    // compile it
    a := self:temp(); // and evaluate it
    if not(IsFinite(a)) then // check for errors
    begin
      a := 0.0;
      err := -48453;
      throw('|evt.ex.div0|, err);
      // call the exception handler (see below)
    end
    else if not(IsNormal(a)) then
    begin
      a := 0.0;
      err := -48452;
      throw('|evt.ex.div0|, err);
    end
    else if IsNan(a) then
    begin
      a := 0.0;
      err := -48451;
      throw('|evt.ex.div0|, err);
    end;
    b := NumberStr(a);
    SetValue(self, 'text, b);
    // set the display to the result
    SetValue(self, 'currentNum, a);
    // and set the numeric value
    SetValue(self, 'isNewEntry, TRUE);
    // and declare it as a complete entry
    SetValue(self, 'operator, "0");
    // reset the operator to a null value
    SetValue(self, 'isFour, TRUE);
  end
  onexception |evt.ex| do
  // the other half of the exception handler
  begin
    self:numExceptHandler(err);
  end;
end
```

A binary function call requires the much simpler `resultview.fourfun`:

```
func(theFun)
begin
  self:evaluator();
  // evaluate any pending operation
  SetValue(self, 'operator, theFun);
  // and set the next binary operation
end
```

The evaluator is the heart for binary operations:

```
resultview.evaluator:
func()
begin
  local b, c, d, e, f, g, cc, ans, err;
  err := -48450; // default error code
  try // wrap in the exception handler
   begin
   SetValue(self, 'lastNum, self.currentNum);
   // push the current number
   g := :StringToNumber2(self.text);
   // get the current displayed number
   SetValue(self, 'currentNum, g);
   // and make its numeric equivalent
   b := self.operator;
   // get the current operator
   if b then
   begin
     c := StrLen(b);
     if (c>0) and (not (StrPos(b, "0", 0))) then
     begin
       d := NumberStr(self.lastNum);
       // now get and clean
       // the current and last numbers
       d := :ThTrimString(d);
       e := NumberStr(self.currentNum);
       e := :ThTrimString(e);
       if (d and e) then
       begin
         if self.isFour then
         // if a binary operator
         begin
           cc := d & b & e;
           // build the Newtonscript string
           self.temp := compile(cc);
           // and compile it
         end
         else // it is a unary operator
         begin
           cc := b & "(" & d & ", " & e & ")";
           // build the string
           self.temp := compile(cc);
           // and compile it
         end;
         ans := self:temp(); // evaluate it
         if not(IsFinite(ans)) then
         // now check for errors
         begin
           ans := 0.0;
           err := -48453;
           throw('|evt.ex.div0|, err);
         end
         else if not(IsNormal(ans)) then
         begin
           ans := 0.0;
           err := -48452;
           throw('|evt.ex.div0|, err);
         end
         else if IsNan(ans) then
         begin
           ans := 0.0;
           err := -48451;
           throw('|evt.ex.div0|, err);
```

```
        end;
        SetValue(self, 'currentNum, ans);
      end;
    end;
  end;
  b := NumberStr(self.currentNum);
  SetValue(self, 'text, b);
  // set the display result
  SetValue(self, 'isNewEntry, TRUE);
  // and reset flags
  SetValue(self, 'lastNum, 0);
 end
 onexception |evt.ex| do
 // the exception handler
  begin
  self:numExceptHandler(err);
 end
end
```

Exception Handling

Main numeric routines are set within exception handlers:

```
try
  begin
  // the code, with any throw() calls necessary
  end
onexception |evt.ex| do
  begin
  // call handler
  end
```

The simple alert-based exception handler looks like this:

```
resultview.numExceptHandler:
func(reason)
begin
  local a;
  if reason = -48450 then
   a := "You tried to divide by 0."
   // set string
  else if reason = -48451 then
   a := "Your number is too large."
  else if reason = -48452 then
   a := "The arguments are out of range."
  else if reason = -48453 then
   a := "The result is infinite."
  else if reason = -48454 then
   a := "You cannot take a log of neg. number."
  else if reason = -48455 then
   a := "The answer is a complex number."
  else a := "You made an unknown error.";
  getRoot():Notify(kNotifyAlert,
   "Scientific Calculator", a);
  // display alert
end
```

String Manipulation

One of the problems with using the compile() approach is that strings passed for compilation must be acceptable to the NewtonScript compiler. Unfortunately, StringToNumber() and NumberStr() exhibit odd and incompatible behavior at times. In particular, NumberStr() inserts (localized) thousands separators, like '5,045,006.74', which choke StringToNumber(), and NumberStr() can suddenly switch to scientific notation (e.g. '4.567e-8') without control, which causes both StringToNumber and the compiler to barf.

The following substitute for StringToNumber copes with scientific notation:

```
resultview.StringToNumber2:
func(theStr)
begin
  local a, b, c, d, e, f, theNum;
  a := StrPos(theStr, "e", 0);
  if a then
  begin
   b := StrLen(theStr);
   c := SubStr(theStr, 0, a);
   d := SubStr(theStr, (a+1), (b - (a+1)));
   e := StringToNumber(c);
   f := StringToNumber(d);
   theNum := Pow(e, f);
  end
  else theNum := StringToNumber(theStr);
  return theNum;
end
```

This next routine strips out thousands separators (of the form stored in self.thousands):

```
resultview.ThTrimString:
func(numStr)
begin
  local a, b, c;
  a := StrPos(numStr, self.thousands, 0);
  while a do
   begin
   b := StrLen(numStr);
   c := SubStr(numStr, 0, a) &
   SubStr(numStr, (a+1), (b - (a+1)));
   numStr := c;
   a := strPos(numStr, self.thousands, 0);
   end;
  if StrPos(numStr, "e", 0) then
   begin
     StrReplace(numStr, "e", "* Pow(10.0, ", 1);
     numStr := numStr & ".0)";
   end;
  return numStr;
end
```

To make thousands separators strippable, it is necessary to determine the exact separator currently in use. This is a piece of user-localized information, which is found using the getLocale() call. For current purposes, when resultview is being set up, it automatically calls the following code, which retrieves this information and stores it in the thousands slot, for use in ThTrimString:

```
resultview.viewSetupFormScript:
func()
begin
  thousands :=
        getLocale().numberformat.groupSepStr;
end
```

Conclusion

This calculator has proved fairly simple to implement, and the executable package is less than 30 K in size. There are some remaining bugs. For instance, the string handling routines are not really set up to handle improperly placed decimal points. If NumberStr() and StringToNumber() were rather better behaved (perhaps, less clever!), this would be much simpler. However, the trick of using Newtonscript compile() and evaluation on the fly appears to be worthwhile and robust. π

The full source code for SciCalc can be found on many shareware bulletin boards, as well as the PIE Developers Source Disk. See the inside front cover for details.

MacsHex- Continued from Page 17

PressOp tells the alert to open when the user attempts a divide by zero, by calling the Open method in the linked subview frame. An alternative would have been to go ahead and try the division, catching the divide by zero in an exception handler. This seemed overkill for a single exception. Similarly, an alternative to the nested if statements would have been to convert the expression to a string by concatenating the values and the operator, compile the string and evaluate the expression. For four functions, this would be only complicate the method. However, a more sophisticated approach would make sense if a larger number of exceptions or functions were possible. You might like to check out Howard Oakley's freeware SciCalc program, which contains an example of both techniques. *[Howad explains this technique in his article starting on page 18].*

The only thing tricky about this application was trying to simulate 32-bit arithmetic. Newton uses 30-bit integers so all the computation had to be done with real numbers which were converted to one to eight digit hexadecimal strings before displaying them. The conversion is accomplished in the methods HexString and DecToHex. HexString converts negative numbers to their positive 32-bit counterparts, since there are no negative hexadecimal numbers. It does this by adding 232 to the negative numbers.

```
HexString
func(dec)
    begin
// dec is in the range -2**32-1..2**31-1
// to display, it must be non-negative
if dec < 0.0 then
 begin
   dec := dec + 4294967296.;
 end;
 return :DecToHex(dec);
end
```

DecToHex is a recursive function which converts numbers into a string representing the equivalent hexadecimal number. The conversion is done by extracting a character from hex (slot value ("0123456789ABCDEF") each time through.

```
DecToHex
func(dec)
begin
  local  digit, char, rest;
  digit := Floor(dec-(Floor(dec/16.0)*16.0));
  char := SubStr(hex, digit, 1);
  rest := Floor(dec/16.0);
  if rest > 0 then  return :DecToHex(rest) &
                    SubStr(hex, digit, 1);
  else return char;
end
```

Conclusion

MacsHex is not rocket science. It doesn't do soups or preferences or advance the state of the art in handwriting recognition, but it is readable and it works. I hope my experience helps demystify Newton development.

The best thing new Newton programmers can do is practice. Use the Newton, get to know it. It is a new platform and behaves in a new way. Hack on the Inspector, poke around the inner workings of the Newton. There are a million things (undocumented) to discover for yourself. It also helps to talk to other programmers. Every day someone discovers something new. I've passed on a few tips based on my first try at a complete application. Ideally, those who have been helped by this early experience will shortly return the favor. π

The full source code for MascHex can be found on many bulletin boards, as well as the PIE Developers Source Disk. See the inside front cover for details.

Chunks

Blowing Smoke

Michael S. Engber

The system view types clParagrah and clEdit automatically handle scrubbing gestures using the ever popular poof sound and dissipating smoke. This article explains how you can use the scrub effect in your own views. This article assumes that the reader has some experience using NTK to write Newton applications. The culmination of this article is a method, SmokeIt, that you define in your application's base view. You pass this method a bounds frame, in global coordinates, describing the rectangle you want "smoked."

Poof Sound

The first problem to solve is how to get the graphics and sound data. Fortunately, we can use the same bitmaps and sound the system uses. The sound is available as ROM_poof and the three bitmaps in the animation sequence are ROM_Cloud1, ROM_Cloud2, and ROM_Cloud3. These pre-defined constants and many others are provided by NTK. In the initial release of NTK, all the constant definitions are in the "GlobalData" text file. In future releases of NTK, the actual definitions will be in the various platform files, but a listing of them can be found in the "NTK Definitions" text file.

Connect the NTK Inspector and execute the following code:

```
PlaySound(ROM_poof);
```

You will hear the familiar poof. We have the sound problem solved. PlaySound plays the sound asynchronously, so all that's remains is to immediately follow the call to PlaySound with animation code.

Cloud Animation

In principle, it sounds easy. Just draw ROM_Cloud1, ROM_Cloud2, and ROM_Cloud3 in rapid succession. I tried a number of different techniques before finding one that closely resembled the way the system does it. Speed was not the problem. In fact, delays need to be inserted between drawing the clouds or they go by too quickly to be seen. We'll start by examining the bitmaps we need to draw. If we use NTK's Inspector to examine ROM_Cloud1, we see the following:

```
{
  mask: <mask, length 2632>,
  bits: <bits, length 2632>,
  bounds: {left: 0, top: 0, right: 178, bottom:
109}}
```

Bitmaps are represented as frames with a bits slot containing the raw bitmap data and bounds slot specifying the rectangle in which to draw the (scaled) bitmap. Bitmaps can optionally have a mask slot containing raw bitmap data that is used to determine which pixels from the bits data are actually drawn. The mask allows you to draw non-rectangular images. Only bits pixels whose corresponding mask pixels are "on" actually get drawn.

Bitmaps are used in three ways in the Newton:

- They can be used in graphic objects that are created with MakeShape and drawn with DrawShape;
- They can be used in the icon slots of clPicture views, in which case the view system takes care of drawing them; and
- They can be passed to the CopyBits drawing function.

Unfortunately, the first two methods won't work for us. when you pass a bitmap to MakeShape the resulting graphic object does not retain the mask data. Since our cloud bitmaps have masks, we can't use MakeShape and DrawShape. clPicture views use the mask data. We could create a dummy clPicture view and animate it by changing its icon slot using SetValue. However, when we closed the dummy clPicture view, the entire view rectangle would be erased instead of just the area covered by clouds. A small point, but it's definitely a different look and feel from the way the system does scrubbing.

CopyBits uses the mask data if you specify modeMask as the transfer mode, and it doesn't require allocating any dummy views. That makes CopyBits the best choice of the three methods, but life is never so easy. We need to be able to draw the clouds scaled. Unlike its namesake in the Macintosh ToolBox, CopyBits only lets you specify the top-left corner of where you want to draw, not the entire rectangle. However, CopyBits does use the bounds slot of the bitmap. We can adjust the bounds slot to achieve suitable scaling. Of course, the three cloud bitmaps are in ROM, so we can't change

them directly. Instead, we clone them and change the bounds of the clones. Below is the relevant code extracted from the SmokeIt function:

```
local bm1 := Clone(ROM_cloud1);
local bm2 := Clone(ROM_cloud2);
local bm3 := Clone(ROM_cloud3);

bm1.bounds := bm2.bounds := bm3.bounds :=
     smokeBounds;
```

Since bitmaps are fairly large objects (we can see that the bits and mask fields occupy 2*2632 = 5264 bytes), you might be concerned that cloning consumes a lot of memory. Remember that we're using Clone, not DeepClone. Clone only copies the top level of a data structure. The only memory allocated is a new, three-slot frame. The bits and mask fields of the clone are still references to the same objects referenced by the original slots.

Below is the code from the single animation step which draws ROM_Cloud1. First we draw the cloud, then we delay a bit, and finally we erase the cloud. These steps are repeated for each cloud.

```
GetRoot():CopyBits(bm1,left,top,modeMask);
Sleep(2);
GetRoot():CopyBits(bm1,left,top,modeBic);
```

Just for completeness, let's consider how the two drawing techniques we abandoned would handle scaling: Graphic objects created with MakeShape can be scaled using the ScaleShape function; clPicture views scale their pictures if you have their content set to full horizontal and full vertical justification.

Forcing A Screen Update

The net result on the screen from each animation step is to erase the area covered by the cloud. After the animation we need to ensure that the screen updates properly. Presumably SmokeIt's caller takes care of making the scrubbed item disappear, by deleting the view for instance. However, smoke clouds usually occupy a larger area than the item being scrubbed. This means SmokeIt must ensure that the entire smokeBounds rectangle gets re-drawn.

If we know that smokeBounds is confined to a single view, we could simply send that view a Dirty message. Since we're writing SmokeIt as a general purpose routine, however, we can't rely on having this information. Alternately, we could send a Dirty message to the root view, but this turns out to be unacceptably slow. What's needed is something like the InvalRect call from the Macintosh ToolBox, but no such call is available.

We can achieve the same effect by creating a dummy view, opening it, and then closing it. Here is the relevant code from SmokeIt:

```
local danQuayle := BuildContext({
                     viewClass:clView,
                     viewFlags:vFloating,
                     viewBounds:smokeBounds});
danQuayle:Open();
danQuayle:Close();
```

BuildContext takes a template as an argument and returns the view it constructed from that template. The view's parent is the root view. The template is not added to the root view's viewChildren array–you can liken the view to a free agent. BuildContext should not be treated as a general purpose method for creating views dynamically. It's only useful in special situations like the one at hand.

The template passed to BuildContext is the bare minimum amount of information you need to create a view. The only part worth discussing is the viewFlags. If vFloating isn't specified, the dummy view appears behind any floating views that happen to be around. Hence, those floating views wouldn't be forced to update when the dummy view is closed.

SmokeIt

The source code to SmokeIt is listed on the nest page. Copy this source into a script slot named SmokeIt in your application's base view. That way it will be available to all the views in your application. The smokeBounds argument is a bounds frame. That is, a frame with a left, top, bottom, and right slot. It needs to be specified in global coordinates. A typical usage might be to delete a view named deadMan. You might do this with code like:

```
:SmokeIt(deadMan:GlobalBox());
deadMan:Close();
```

In practice, you'll find that you want the rectangle to be somewhat larger than the item being deleted so that the clouds completely obscure it. I find that expanding 25% in each direction is adequate.

Some useful functions and methods for creating bounds frames are SetBounds, RelBounds, StrokeBounds, :GlobalBox, and :LocalBox. They are all documented in the *Newton Programmer's Guide.*

```
func(smokeBounds)
begin

 local top := smokeBounds.top;
 local left := smokeBounds.left;

 local bm1 := Clone(ROM_cloud1);
 local bm2 := Clone(ROM_cloud2);
 local bm3 := Clone(ROM_cloud3);

 //CopyBits draws bitmaps scaled to their bounds
 bm1.bounds := bm2.bounds := bm3.bounds :=
smokeBounds;

 //explained below
 local danQuayle := BuildContext({
           viewClass:clView,
           viewFlags:vFloating,
           viewBounds:smokeBounds});

 PlaySound(ROM_poof);

 GetRoot():CopyBits(bm1,left,top,modeMask);
 Sleep(2);
 GetRoot():CopyBits(bm1,left,top,modeBic);

 GetRoot():CopyBits(bm2,left,top,modeMask);
 Sleep(2);
 GetRoot():CopyBits(bm2,left,top,modeBic);

 GetRoot():CopyBits(bm3,left,top,modeMask);
 Sleep(1);
 GetRoot():CopyBits(bm3,left,top,modeBic);

// force update (a la InvalRect in Mac ToolBox)
// If you knew what view(s) to dirty you could
// just use :Dirty() to force the update. This
// code is general purpose, so it doesn't
// know (and GetRoot():Dirty() is way slow.)

 danQuayle:Open();
 danQuayle:Close();

end
```

π

Chunks

Downloading Data To Your Newton

George W. P. Henne
The Eastwood Group
72714.3043@compuserve.com

One of the first roadblocks I ran into developing for the Newton was finding a way to download some data to help build a data table for an application. With the current documentation, it wasn't easy. The answer is to use a few statements not documented in the Alpha NTK manual, and the proper technique.

Downloading

Before following the steps outline below, you should first put your data into a printable text file–it's easiest to deal with that format on the Newton. Then, using the following steps, you should be able to download your data.

1. Using ResEdit, create a resource file named DATA.rsrc (or any name of your choice). Add a resource of type RTEXTS. Paste into it the data you want to download . Make sure you name your resource: you'll need the name to access it from NTK. In our example, we'll use "MyData".

2. In NTK, use Project...Add File... to add the name of the resource file DATA.rsrc to the package. It can now be accessed by the code in Project...Project Data.

Project Data is not well documented in the initial version of the NTK docs. I have the feeling there's a lot more power and functionality there than we've seen so far. Project Data contains code which is executed, on the Mac, at compile time just before the download. It seems to support the same NewtonScript language that runs on the Newton. I'm not sure how complete the implementation is, but it makes one wonder whether a version of NewtonScript for the Mac is a possibility.

3. Add the following code into Project...Project Data (you can't have any views open to edit Project Data):

```
OpenResFile(":Newton Apps Folder:DATA.rsrc");
gData:=GetNamedResource("TEXT","MyData",'string);
CloseResFile();
```

OpenResFile and CloseResFile are undocumented in the original NTK docs. It's important to remember that the pathname for OpenResFile starts from the NTK's root directory (in this case, my Newton Apps Folder is within the NTK folder). Only one resource file can be open at a time. To open more than one resource file at a time, use OpenResFileX(filename). It returns a file number, which can be used by a subsequent CloseResFileX(file number).

The GetNamedResource function is a powerful one. The arguments are, in turn, the resource type, the resource name, and the data class. The resource that is named is converted to a binary object of the data class. It also works for other types of resources: I'll give some examples later. Apple warns that the method for opening and closing resources will change in the future in Project Data. I believe the method I'm using will still be valid, however.

4. To your main view, add an evaluate slot named "MyData". Its value should be the name of the slot to which you assigned your data (in this example, gdata).

5. Once you've done a build and a download, you can then access the data using the ExtractChar() function. This function is documented, but it's not clear what it's for from the documents. Here's how to use it:

```
for i:=1 to 40 do
begin;
  x:=extractChar(MyData,i);
  print (x);
end;
```

Text data in the Newton is stored as Unicode. Each character is represented by two bytes. The incoming data is Macintosh text with one byte per character. In the above code, each character (it assumes the incoming string is 40 bytes long) is extracted into the slot x, which contains a two byte Unicode representation. The print(x) statement displays this character to the Inspector window: most likely, you will want to do some manipulation before storing it in your soup. Note that it you attempt to print(MyData), you get a bunch of blocks on the screen. Macintosh text is unprintable.

Downloading in the Real World

This method works quite well for me: I use the method in several ways. The application I was developing was originally written in HyperCard, and had some data tables which drove the application. Without using this method, I would have had to enter the data directly into the Newton by tapping on its keyboard, a time consuming and (for me) inaccurate process. Alternatively, I could have hard coded the data in Project Data as statements, also cumbersome.

There are other types of data that can be moved in a similar fashion. For example, to move a number of PICT resources, do the following:

```
OpenResFile(":Newton Apps Folder:DATA.rsrc");
gMaps:=[];
AddArraySlot (gMaps, {name: "France", map:
   GetNamedResource("PICT","France",'picture)});
AddArraySlot (gMaps, {name: "Calif", map:
   GetNamedResource("PICT","Calif",'picture)});
CloseResFile();
```

Here I extend our use of Project Data in another way. Rather than storing my object as a simple slot, I set up an array. Once I'm running on the Newton, I access the array in much the same manner as the TEXT example. (I have an evaluate slot named Maps in the base application with the value gMaps. The slot Maps contains an array of maps I can use throughout the application.) As you might expect, the PICT resources are added to the resource file the same way I loaded in the TEXT resource. No transformation is needed on a PICT resource once it's on the Newton: the objects are immediately usable.

How cumbersome is this in practice? Not too bad, it turns out. I download about 35K of resources with my package, and it makes little difference to the compile or download times (at most a few seconds). I'm more concerned with the extra space consumed by two versions of the text: one as binary, the other as Unicode. To avoid this problem I split my application into two parts: an installer, which has the Project Data and creates soups with the downloaded data, and the actual application, which then makes use of the soup. The installer can be discarded after it's done its thing.

There are a few variants that I don't describe in this article that you should be aware of, should you need them, including GetPICTasBits(name, flag), GetSound(name) and GetSound11(name). You need to do some research before you can use them effectively: there's probably another whole article somewhere here.

I'd like to thank all those who gave me clues to developing this method. I got bits and pieces from a number of sources, and was understandably thrilled when they all came together. π

George Henne is with The Eastwood Group, specializing in the application of new technologies. You can reach him at (416) 264-5777, fax (416) 264-5888, or CompuServe: 72714,3043.

Soup's On

Michael S. Engber
Apple Computer - PIE Technical Support

Serious Business

With the myriad of different soup and store functions available on the Newton, it's hard to tell which functions are important and how they should be used. This article discusses how an application should properly create and use soups. This article is not a tutorial on the topic of soups. This article assumes that the reader is familiar with the chapter on soups in the Newton Programmer's Guide and has some experience using NTK to write Newton applications.

Union Soups

The first rule of using soups on the Newton is "Use union-soups." Union-soups handle the details of whether you should be writing data to the internal store or an external store, a PCMCIA card. The Newton user should be in control of where his/her data goes. When the user inserts a card or taps the Card item in the extras drawer, the Card Storage dialog comes up. This has a checkbox titled "Store new items on card" with which the user controls where data goes. By using union-soups your program automatically respects the user's wishes.

You can think of union-soups as grouping together the soups with a given name on the various available stores. Currently, that is just one or two soups, but future Newtons may change that. Later in this article I discuss how to properly create union-soups.

There are three basic calls that should handle 99% of your soup related code. First, there's GetUnionSoup. It takes a soup name (a string) as its parameter and returns a union-soup. Next there's the union-soup method, AddToDefaultStore, which does exactly what its name says. It stores a new entry to the store the user has chosen as the default. And finally, there's EntryChange, which you use to make sure any changes you make to a soup entry are written out to the storage device.

Stores and Soups

Stores can be likened to volumes and soups can be likened to databases. A store can contain many soups just as a volume can contain many databases. A soup is comprised of entries just like a database is composed of records. Soups can maintain indexes on their entries. These indexes can be specified when the soup is created or added or deleted later.

There are a variety of soup and store methods. They are documented and in practice you rarely need to use them, so I won't say much about them here. The GetStores() global function returns an array of the available stores. You can assume that the first element of the array is Newton's internal store. Currently, the second element of the array, if it exists, is always the PCMCIA card, but this assumption won't necessarily hold for future Newtons. Consider the problem of a Newton with multiple slots.

When debugging from NTK's Inspector it's sometimes useful to use the GetSoupNames method. For instance, `GetStores()[0]:GetSoupNames()` returns an array of the soup names from the internal store.

Entries

Soups are comprised of entries. As you access the soup, frames are created for the entries. Normally, you can just think of soups as consisting of these frames. In reality, these frames are just caching information from the soup into RAM. That's why it's necessary to call EntryChange to write the information from an entry's frame in memory back out to the soup. Otherwise, the change would be lost when the Newton is restarted.

Deciding when to call EntryChange is sometimes difficult. For instance, if you have an input line, you wouldn't want to call EntryChange from its viewChangedScript. If the user was entering data into the line with the keyboard, your viewChangedScript gets called after every key press. Calling EntryChange with every key press would be noticeably slow.

In some situations it's obvious when to call EntryChange. For example, a natural time to call EntryChange is when the user dismisses a dialog box. Other cases may not be so clear cut. The Notepad uses a viewIdleScript to call EntryChange about every five seconds. That's why the Newton user manual suggests waiting 5-10 seconds before removing a PCMCIA card or you may lose data. If you manage to corrupt the Notepad or its soup while hacking around, you may get error dialogs every five seconds, each time the Notepad tries to write to its soup.

There isn't much structure imposed on soup entries. For instance, there's no requirement they all have the same slots. All entries have a _uniqueId slot which contains an integer uniquely identifying the entry within its soup. The system creates this slot for you, so you don't need to worry about it. In fact, the _uniqueId slot is only unique within a soup, not within a union-soup.

Since applications normally use union-soups, the _uniqueId slot isn't of much practical use.

You're free to put any slots you want in your entries and to vary them from entry to entry. The only slots you have to worry about are the slots that are indexed. When you create a soup, you decide what slots are used to build indexes. If you specify that an index should be built on slot foo and that foo contains a string, then it's important that you make sure that the foo slot in every entry contains a string. It's permissible for an entry not to have a foo slot at all, in which case it won't participate in queries on the foo index. The essential point is that if an entry has an indexed slot, the slot contains data of the right type.

To create an entry, you should create a new frame and pass it as the parameter to an AddToDefaultStore message. The frame is destructively modified. You'll see that it now has a _uniqueId slot. When a frame is added to a soup, a deep copy of it is made. That is, all its slots are followed and everything those slots reference is followed and copied, and so on, until everything is copied into the soup. This is necessary because soups persist across restarts, so they can't contain references to objects in RAM. Because of this you need to think carefully about the data structures you use for soup entries. For instance, a view whose _parent chain ultimately leads to the root view would be a very bad candidate for a soup entry.

There are two exceptions to this copying rule. Pointers into ROM are not followed. Since the ROM is persistent across restarts, pointers into ROM are safe to save in soups. The second exception is _proto slots. A frame's _proto slot is not followed when creating a soup entry, nor is a _proto slot created for the entry. The _proto slot is not removed when a frame is added to a soup, so it may appear that you have a soup entry with a _proto slot. However, the frame is only a temporary cache of the entry's data. Next time you retrieve the entry from the soup, you see it does not have a _proto slot. If you need it, you will to create it every time you retrieve the entry.

Cursors

Cursors are used to navigate through soups. You obtain a cursor as the result of a call to Query. This simplest possible query is:

```
curs := query(uSoup,{type: 'index})
```

I use this technique in the NTK Inspector to get a "quick and dirty" cursor for peeking at entries in a soup. I say "quick and dirty" because this code is sloppy. The argument specifies a query type of index, but it doesn't specify an indexPath. This works because all soups have an index on the _uniqueId slot which is used as the default indexPath.

There are a wide variety of options for specifying queries, as well as a wide variety of cursor functions. They are all documented in the *Newton Programmer's Guide*, so I won't go into them here. Here are a few points about using cursors.

- Cursor:Entry() can return nil if the cursor runs off either end of the soup. It can return the symbol 'deleted if the entry it references was deleted. Be sure your code can handle these cases.
- Traversing your soup creates a frame for each soup entry as it is encountered. (Yes, these frames are cached, so you don't pay a penalty for visiting the same entry twice.) Avoid traversing every entry in the soup. Use indexes and startKeys to limit your search. Using validTests doesn't help with this problem. The entry frame has to be created in order to pass it to the validTest.
- Don't leave unnecessary references to entries lying around. For instance, building up an array of entries ties up a lot of memory. As soon as you're done with the array, delete any references to it so it can be garbage collected, which in turn allows all the entry frames it contains to be garbage collected.

Creating Your Application's Soup

The benefits of using a union-soup depend on one crucial point we've overlooked so far. The individual soups which comprise a union-soup must exist on all available stores so the user can direct his/her data there. There is a global array, CardSoups, which is used by the system to handle this. The elements of CardSoups are used in pairs. The first element of the pair is the soup name, a string. The second element is an array of index specifications (the frames you use to specify indexes when creating a soup). Whenever a card is inserted, the system uses this array to make sure each soup in CardSoups exists on the card, creating soups as necessary.

The first rule of using soups on the Newton is "Use union-soups."

You need to use the function CreateAppSoup when creating soups. Call CreateAppSoup to make sure your soup exists on the internal store. CreateAppSoup creates it only if necessary. CreateAppSoup takes four arguments: the soup name, an array of soup indices, a one element array containing your application's appSymbol, and your application's appObject. (appObject is an array of two strings, such as ["entry", "entries"], that describes the singular and plural of the data in that soup. Filing expects the appObject slot of your base view to contain an array like this.) The first two argument are used to create the soup; see CreateSoup's documentation for all the details on index specs. The last two arguments are added to the soup's info frame as an annotation describing who uses the soup and what's in it. Due to a bug in the SetInfo soup method, you need to use EnsureInternal to make sure these second two arguments reside in internal memory.

There are conventions for naming your soup. If your application creates only one soup, you should use your package name for the soup name. Your package name is formed by concatenating your Extras drawer name with your unique signature, for instance "foo:myCompany". If your application creates multiple soups, use your package name as a suffix to a descriptive soup name ("soup1:foo:myCompany").

Basic Methods

Putting all this information together into a simple recipe, I recommend you define two methods in your application's base view, RegisterCardSoup and UnRegisterCardSoup. Here is their code.

A view whose _parent chain ultimately leads to the root view would be a very bad candidate for a soup entry.

```
//constants defined in "Project Data"
constant kAppSymbol := '|llama:PIEDTS|;
constant kPackageName := "llama:PIEDTS";
constant kAppObject := '["llama","llamen"];

constant kSoupName := kPackageName;
constant kSoupIndexes := '[{structure: slot,
      path: name.first, type: string}];

//2 base view methods

RegisterCardSoup:
func(soupName,soupIndexes,appSymbol,appObject)
 //returns a union-soup for your app to use
 begin
  //first check for system provided function
  if functions.RegisterCardSoup then
   return RegisterCardSoup(soupName,soupIndexes,
                           appSymbol,appObject);
  CreateAppSoup (soupName, soupIndexes,
            EnsureInternal([appSymbol]),
            EnsureInternal(appObject));

  //ensure your soup will exist on stores
  //which later become available
  AddArraySlot(CardSoups,soupName);
  AddArraySlot(CardSoups,soupIndexes);

  //ensure your soup exists on all
  //currently available stores
  local store;
  foreach store in GetStores() do
   if NOT store:IsReadOnly() AND NOT
         store:HasSoup(soupName) then
    store:CreateSoup(soupName,soupIndexes);
  GetUnionSoup(soupName);
 end,

UnRegisterCardSoup: func(soupName)
 begin
  //first check for system provided function
  if functions.UnRegisterCardSoup then
   return UnRegisterCardSoup(soupName);

  local pos := ArrayPos(CardSoups,soupName,0,
     func(x,y) ClassOf(y)='String AND
     StrEqual(x,y));
  if pos then ArrayRemoveCount(CardSoups,pos,2);
 end;

//in your viewSetUpFormScript, call
:RegisterCardSoup(kSoupName,kSoupIndexes,
     kAppSymbol,kAppObject);

//in your viewQuitScript, call
:UnRegisterCardSoup(kSoupName);
```

For now, you should define RegisterCardSoup and UnRegisterCardSoup as methods in your base view. In future Newtons, these functions may be provided in the ROM. The methods defined above allow your application to take advantage of this future functionality by first checking to see if these global functions are available.

Your application should call RegisterCardSoup when it starts up. RegisterCardSoup works by first making sure your soup exists in the internal store. Next, it registers your soup with the CardSoups global variable. Then it ensures your soup exists on any cards already inserted. Finally, it calls GetUnionSoup and returns that value. Your application can use the value returned by RegisterCardSoup as an alternative to calling GetUnionSoup directly.

Your application should call UnRegisterCardSoup when it quits. UnRegisterCardSoup simply unregisters your application from the CardSoups global variable. Your application only needs to worry about its soup existing on all available stores while it's actually open. If a new card is introduced while your application is closed, it is handled automatically when your application gets opened–RegisterCardSoup is run and creates the soup. When your application quits, it should also take care to remove any references it has to entries, soups, or cursors. This usually means setting to nil any slots in which you stored soup or cursor references.

Sharing Soups With Others

Newton programmers are encouraged to reuse existing soups for their own needs. The format of the standard soups is documented in the Newton Programmer's Guide. This section discusses how to properly handle the sharing of data.

Registering With SoupNotify

Applications can be notified when a particular soup gets changed. For instance, maybe your application has developed a following of third party utilities which manipulate your soups, or more likely, you're using a built in soup, like "Names." If you implement the Find feature, the FindAll overview notifies your application of changes it makes to your soup using this same mechanism. It's generally a good idea for your application to set itself up for notification.

To register for notification all you have to do is add two entries to the SoupNotify global array. Your application should register in its viewSetupDoneScript and unregister in its viewQuitScript. The elements of SoupNotify are used in pairs. The first element in the pair is the name of the soup. The second element is the appSymbol of the application to be notified. Below is some code to register an application for changes to the System Soup. This code assumes that a constant, kAppSymbol, has been defined for the application's appSymbol:

```
//register w/ SoupNotify in viewSetupDoneScript
AddArraySlot(SoupNotify,ROM_SystemSoupName);
AddArraySlot(SoupNotify,kAppSymbol);

//unregister with SoupNotify in viewQuitScript

local soupNotifyPos := ArrayPos(soupNotify,
      kAppSymbol,0,nil);
ArrayRemoveCount(soupNotify,soupNotifyPos-1,2);
```

Notice that the code for unregistering looks for the appSymbol instead of the soup name. This is essential since other applications may have requested notification for the same soup. Also note that this code can easily be generalized to unregister you from multiple soups:

```
//unreg all SoupNotify entries in viewQuitScript

local soupNotifyPos;
while soupNotifyPos := ArrayPos(soupNotify,
      kAppSymbol,0,nil) do
 ArrayRemoveCount(soupNotify,soupNotifyPos-1,2);
```

Receiving Notifications

Once your application is registered it receives soupChanged messages when soups change. You need to define a soupChanged method in your base view which accepts one argument, the name of the soup that changed. Your application should respond to this in whatever manner is appropriate. Normal applications have no need to respond unless they're open. That's why I recommend you register when your application opens and unregister when your application closes.

Traversing your soup creates a frame for each soup entry as it is encountered. Avoid traversing every entry in the soup.

Sending Notifications

If your application makes changes to a shared soup, it should use the BroadcastSoupChange function to notify other applications of the change. BroadcastSoupChange takes a single argument, the name of the soup that changed. Unfortunately, the current version of the Prefs application doesn't use BroadcastSoupChange. So if you try to register with the System Soup, don't be surprised if Prefs doesn't notify you.

Making Changes to Other Application's Soups

In addition to using BroadcastSoupChange, there is another convention you should following when modifying soups not owned by your application. If your application needs to add its own slots to entries in another application's soup, you should create only a single slot. Use your appSymbol as the slot name. This avoids name conflicts, keeps the entries from getting cluttered up, and allows for easy removal of your data.

Cleaning Up Spilled Soup

You may have been wondering "Doesn't this mean that while my application is open, it creates soups on every card inserted into the Newton?". The answer is "yes." However, the soups will be empty unless the card is the default store and the user creates data with your application.

You may also have been wondering "If the user inserts lots of cards in the process of using my application, can't his/her data be spread out all over the place?". Again, the answer is "yes." There's not much you can do about this. Hopefully Newton users will learn to organize their data on cards, just like computer users learn organize their data on floppy disks.

The issue I haven't discussed is deleting your soups. Similar issues are deleting your preferences from the System Soup and deleting modifications you've made to other applications' soups. Analogous problems exist in the Macintosh world. How do you clean up your Preferences folder or get those out of date icons out of the Desktop Database?

I don't have a good answer. You certainly don't want to do it in your viewQuitScript. You probably don't want to do it in your RemoveScript. Remember, your RemoveScript is run when the card containing your application is removed, as well as when the user chooses Remove Software from the Prefs or Card applications. There is no straightforward way to determine which event is happening.

I suspect the eventual outcome will be one, or more, of the following:

- A reliable method to distinguish having your card removed and Remove Software will be documented. Then, when your application detects the Remove Software case, it can prompt the user to see if they really want all traces of the application removed. You wouldn't want to do this without prompting the user. Suppose the user is simply deleting an extra copy of your software from a card.
- Users will explicitly do the cleanup. Some applications will provide this option to users–as a button, part of their Prefs panel, or as a utility application. Alternately, if applications follow the guidelines for naming their soups and preferences data, it should be possible to write a cleanup application.
- Chaos will reign. Soups will build up until the ozone layer is depleted. Life, as we know it, will cease.

Preferences–Exception to Always Using Union Soups

You may recall that I said applications should always use union-soups. There's one exception–the SystemSoup, where preferences are stored. It doesn't make much sense to have your preferences spread out on PCMCIA cards. If you know your application is always run from a particular card, it might make sense, but in general, using the internal store seems to be the right thing.

Since everyone has to share the SystemSoup and the Newton's internal store, be considerate and don't keep large amounts of data there. Below is some code that retrieves this preferences information and creates it if necessary.

```
local sysSoup := GetStores()[0]:GetSoup
     (ROM_SystemSoupName);
local cursor := Query(sysSoup,{type: 'index,
     indexPath: 'tag, startKey: PackageName});
local prefsEntry := cursor:Entry();
if NOT (prefsEntry AND
  StrEqual(prefsEntry.tag,kPackageName)) then
  prefsEntry := sysSoup:Add({tag: PackageName,
     appSpecificSlot: nil});
```

Conclusion

Your application should allow the user flexibility in storing data. The user might keep your application internally and store its data on PCMCIA cards (even if your application is too big to fit in the internal store of the current Newton, don't assume this will be the case for future Newtons.) Alternately, your application might be kept on a PCMCIA card along with its data. On a two slot Newton, your application might be stored on one PCMCIA card and its data on another. You should design your application with enough flexibility to handle any configuration. Union-soups provide an easy way to do this. π

Tales From The View System

Serious Business

Michael S. Engber
Apple Computer, PIE Technical Support

This article presents details about the Newton's view system that are often overlooked or misunderstood. You can write significant Newton applications without knowing all these details (largely due to NTK's ease of use), but ultimately, you need to understand many of the topics presented here.

This is not an introductory article. It's designed to be read on your second or third pass through understanding the Newton's view system. Don't expect to understand everything it contains the first time you read it. This article assumes that the reader knows the basics of NewtonScript (especially proto and parent inheritance) and has some experience using NTK to write Newton applications.

Templates vs. Views

Views are not created by NTK. In NTK you lay out templates. Views are frames created in RAM by the Newton's view system. Views use templates as their prototypes. The difference between views and templates seems obvious enough, but in practice, it's easy to get confused. Figure 1 illustrates the relationship between a typical view, template, and prototype.

Note several things about Figure 1:

- The view has very few slots, just _parent, _proto, and viewCObject. Views inherit most of the slots and methods they need. This lets views remain small, which is important since they take up space in RAM. Note: if the view has declared children, it would contain a slot for each declared child. Details regarding these slots are discussed later in this article.
- Figure 1 shows why the slot assignment rules work the way they do. Recall, if you change the value of a slot inherited from the _proto chain, a new slot is created in the frame to hold the new value (rather than modifying the inherited slot). If we look at the figure, we can see that only the view is in RAM. Modifying the inherited slot is not even possible. So, even though a view frame starts out very small, it grows over time as slots get modified. This growth is usually very small, since in practice, views seldom ever modify more than one or two slots.
- The template does not have a _parent slot.This might be surprising since NTK displays templates using a parent-child hierarchy. NTK actually uses the hierarchy which exists among the view objects created from the templates.

Table 1 on the next page is a quiz to test your understanding of the distinction between views and templates.

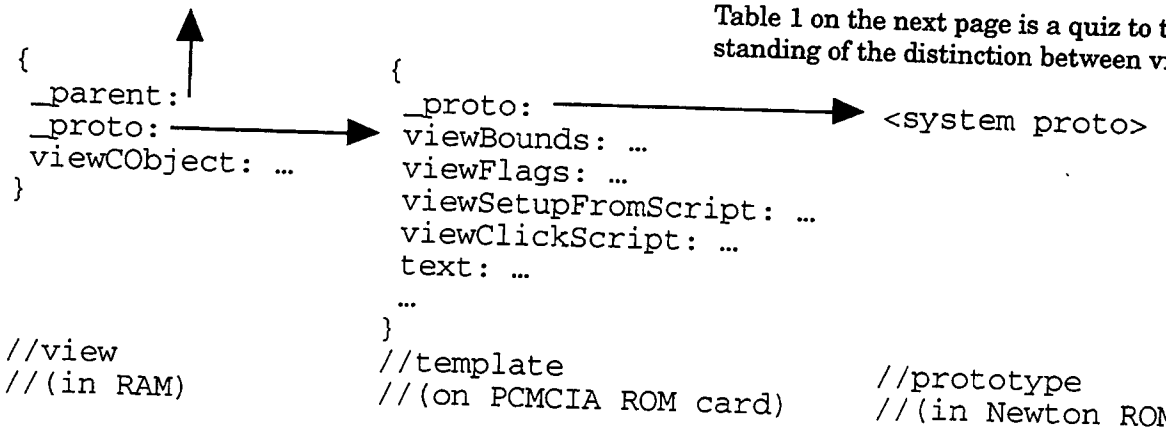


Figure 1 - A typical view/template/proto relationship.

Templates vs. Prototypes

The distinction between templates and prototypes is a bit subtle. I could even argue that, technically, there is no difference. However, it's useful to draw the distinction, especially since NTK enforces this distinction. The things you drag out in NTK are templates. The NTK palette that you make selections from contains prototypes (well, viewClasses too, but we won't go into that now). In general, templates contain prototypes in their _proto slot.

viewChildren and stepChildren

The viewChildren and stepChildren slots both contain template arrays. These arrays are used to create a view's children when the parent view is first created. In general, you use stepChildren for children you define, and the system uses viewChildren for its children. For example, if you create a template based on ProtoFloatNGo, the close box is provided by the system and its template is in the viewChildren array. The templates you add are put into the stepChildren array by NTK. Normally, you need not concern yourself with these details, unless you're explicitly defining a template as a NewtonScript frame rather than dragging it out graphically from the palette.

Always remember: "viewChildren and stepChildren contain templates, not views." Dereferencing some unsuspecting element of your stepChildren or viewChildren array and sending it a Hide message does absolutely no good. It generates an error. If you need access to the views that are created from your viewChildren and stepChildren templates, use the ChildViewFrames message. It returns an array of all the views created from *both* the viewChildren and stepChildren. The viewChildren views are first, followed by the stepChildren views in sibling order (the order they show up in NTK browser). In the FloatNGo example described earlier, ChildViewFrames returns an array whose first element is a close-box view, followed by all the views for the items you define.

Why viewChildren and stepChildren?

If there is only one slot, say the viewChildren slot, then your template's viewChildren slot overrides the viewChildren slot of your template's proto. Returning to the ProtoFloatNGo example, the close box would be missing. When the view system instantiates the view, it only sees the items you define and in this case misses the close box.

This problem can be resolved by searching the entire proto chain for viewChildren slots when creating a view. But this is inefficient. Instead, a parallel slot is used. It's unfortunate that the term stepChildren makes the stepChildren slot sound inferior in some way. The only difference between stepChildren and viewChildren is that the views created from viewChildren are first in the ChildViewFrames array.

Note: At first glance this may appear to be a problem for user protos–you might expect the stepChildren slot in your template to override the stepChildren slot in the userProto it's based on. This isn't a problem because NTK combines the stepChildren of the entire proto chain when it compiles your template. This process doesn't occur until you build your package, so you're normally unaware of it. If you look at your view's stepChildren slot in the inspector, however, you will see all your template's children plus any children defined in its proto.

It may seem possible to combine all the stepChildren and viewChildren at compile time and use one slot to reference all of them. This is not possible–the viewChildren are defined in templates in the Newton ROM and aren't available at compile time.

Table 1 - A template vs. view quiz.

		template	view
1)	created by NTK		
2)	only reside in RAM		
3)	handles `:Hide()`, `:Open()`, and `:Close()` messages		
4)	reside in stepChildren and viewChildren arrays		
5)	code modifies their slots at runtime - preferably, using `SetValue`		
6)	use system protos in their proto slots		
7)	passed to AddStepView		
8)	usually contains few slots (inherits, most of its slots)		
9)	usually reside in ROM		
10)	returned by `:ChildViewFrames()`		
11)	returned by Debug()		
12)	is a frame		
13)	has a parent		

Declaring Views

Declaring a view allows you to access it symbolically. For example, if you declare a view SomeDeclaredView then you can write code like `SomeDeclaredView:Open()` to display it. In NTK you declare views using the Template Info... command. Both the view being declared and the view to which it's being declared must be named. You select the view you want to declare and execute the Template Info... command (Special menu). Figure 2 shows the dialog which appears. You name the view using the edit text field, and use the Allow Access From checkbox and popup to declare the view. Only named parent views appear in the pop up.

What Declaring Really Does

Declaring a view does several things. First, it creates a slot in the parent view. The slot's name is the name of the declared view and the slot's value points to the new view. Because of this it's a bad idea to pick view names like _proto or viewClickScript. Deciding where to declare a view amounts to deciding where you want that view's slot to go.

Remember, NTK only deals with templates. Views are declared. NTK sets up the templates with the correct information (like their names) needed for the declarations. The view system actually takes care of the work when the views are created. The details of this process are discussed in more detail later in this article.

Where to Declare a View

Consider the view hierarchy shown in Figure 3. If we declare view3 in view2, then scripts in view2, view3, and view4 have access to view3 (since the slot is in view2). View3 and view4 get access through parent inheritance (scripts in view3 could also use self) –view1 is left out in the cold. If we declare view3 to view1, instead, everyone has access. Alternately, if view2 is declared to view1 then view1 can access view3 using `view2.view3`.

You should declare a view to a common ancestor of all the views that need access to it. You can simply declare everyone to your application's base view (everyone's ancestor), but you may not want to do that because of potential name conflicts, efficiency considerations (to be discussed later), and most importantly, because it entirely eliminates the challenge of deciding where to declare a view.

Why is Declaring Necessary

On the surface, declare seems just like a convenient way to create a view slot containing self. Couldn't we achieve the same effect by simply having each view we want declared do something like:

```
:Parent().myName := self;
```

in its viewSetupDoneScript? In addition to allowing symbolic access to a view, declaring a view also ensures that it is preallocated (the view frame consisting of the _parent, _proto, and viewCObject is made ahead of time). This means that declare has some overhead associated with it–not a lot, but you should only declare the views that you need to access.

Note that invisible views are never created unless they are declared. This makes sense if you think about it–if they aren't declared there is nowhere to send the Open (or Show) message. The declare mechanism is necessary to handle the case of initially invisible views like dialog boxes. If the declared view is initially visible then you can write code in its viewSetupDoneScript to achieve the desired effect. The decision is up to you. Most people find it easier to just to use the declare mechanism.

Preallocation

It's important to understand the difference between preallocating and opening a view. A preallocated view has its _parent and _proto slots set up and its viewCObject slot set to nil. This actually uses very little RAM. When the view is opened, a more substantial object is allocated in RAM and viewCObject is set to reference it. You can check if a view is opened by checking if its viewCObject slot is non-nil.

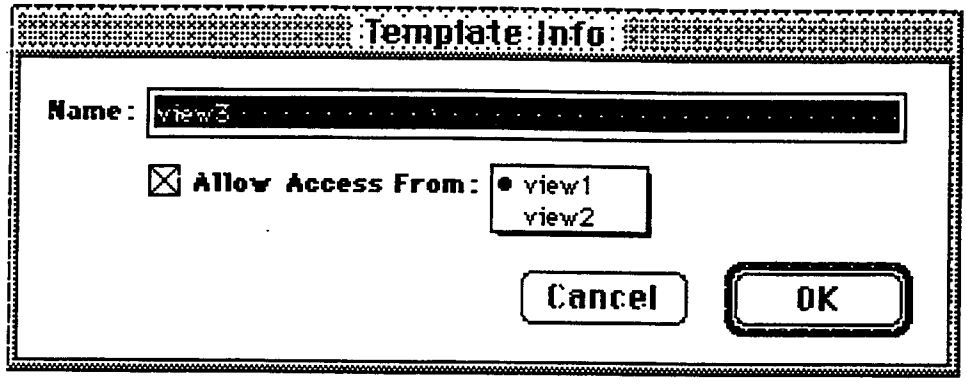


Figure 2 - The NTK's Template Info dialog.

When a preallocated context is initially created, its _parent slot is set to reference the view to which it's declared. This may or may not be its true parent (it may be a grandparent). When it's time to use the preallocated context to open the declared view, the _parent slot is changed to the true parent. Since this happens before the declared view's viewSetupFormScript is executed, you're normally unaware of all this. There is one case, however, where this can affect you. If the declared view is initially invisible it doesn't get opened automatically–you have to explicitly open it. Its _parent slot doesn't get updated automatically. This problem rarely occurs in practice. Most views are either declared to their parent or are initially visible.

Declare Implementation Details

In order to accomplish pre-allocation, declaring a view creates some extra slots in both the template being declared and the template to which the view is declared. In the template being declared, a preallocatedContext slot is created which contains the name you gave it in NTK. Before a view is created from a template, the view system first checks for a preallocatedContext slot in the template. If the slot exists, then this view has been declared (and allocated)–this preallocated view is used instead of creating a new one. The symbol in the preallocatedContext slot is used to find the preallocated view–the parent chain is searched for a slot with this name.

A template with declared views has a slot for each declared view. These slots' names are the names of the views being declared. They are initialized to nil.

A template with declared views also has a stepAllocateContext slot. This slot contains an array with two entries per declared view. The two entries are the declared view's name and a reference to the declared view's template. In system-defined templates and protos there is a similar allocateContext slot. The stepAllocateContext slot is for declared stepChildren, the allocateContext slot is for declared viewChildren.

The stepAllocateContext and allocateContext slots are used when a view is opened, in order to pre-allocate all the declared subviews. For each pair of entries in these arrays, there is a slot in the view. The slot's name (the subview's declared name) is the first element of the pair. The slot contains a preallocated view.

To try and help you visualize all these details, consider the simple case of two views, view1 and view2, with view2 declared to view1. Figure 4a shows the visual hierarchy; Figure 4b shows some of the underlying data structures.These details are rarely of much practical use, but if you're crawling around in the Inspector and see these weird slots in your templates, you'll know they're part of the declare mechanism.

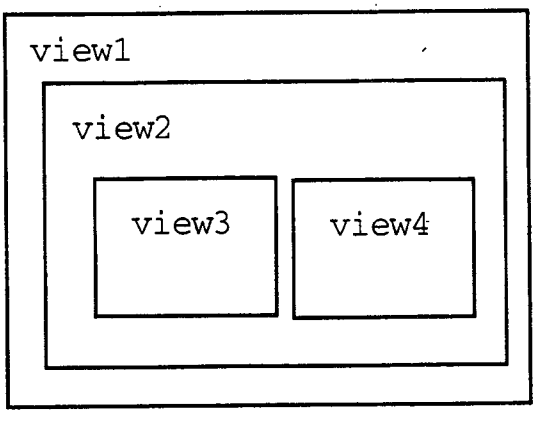


Figure 3 - An example view hierarchy.

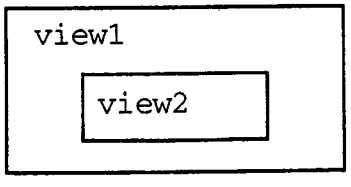


Figure 4a - A simple declaration.

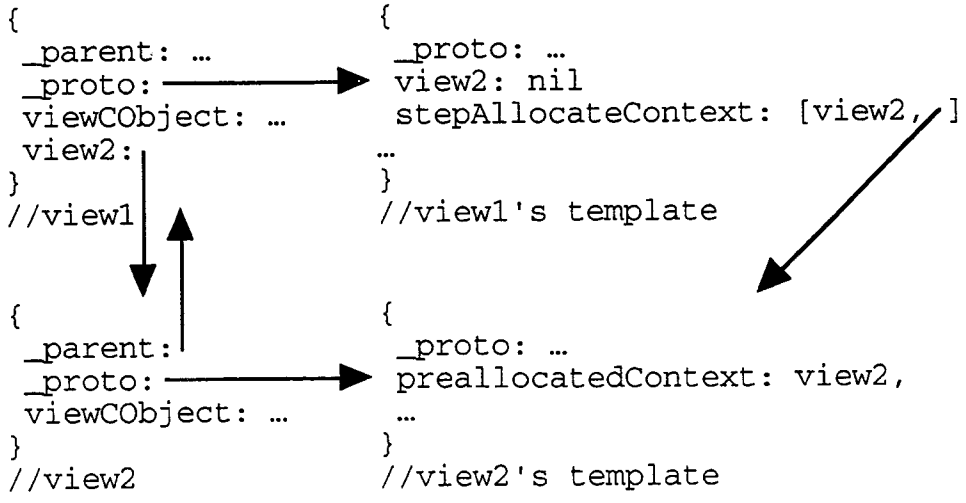


Figure 4b - The data structures underlying Figure 4a.

View Opening and Closing

When a view is opened, all of its declared subviews are preallocated. When a view is closed, all those slots containing the preallocated views are set to nil. Declare keeps pre-allocation exactly one step ahead of you. At any point in time, all the open views have their declared children preallocated, ready to be opened. Closed views can't do anything anyway, so setting their pre-allocation slots to nil allows garbage collection.

When a view is closed, the slots containing the preallocated contexts are set to nil instead of being deleted. For your application's base view, this has an interesting implication. The first time your base view is opened, a slot for each declared view is created. When your base view is closed, those slots are not deleted. If you're looking for memory leaks by comparing free memory before you open and after you close your application, you'll see a slight discrepancy the first time you open and close it due to these new slots. This shouldn't be a problem on subsequent opens and closes.

Pre-allocation happens early in the view opening process, before the viewSetupFormScript is called. When it comes time to instantiate subviews, if the subview has a preallocated context, that preallocated context is used, otherwise a new one is created. Because pre-allocation occurs so early in the view instantiation process, there is a subtle bug that can arise in your program. If you declare a view (let's call it X), and send it an Open message in your viewSetupFormScript, it already has its visible flag set–two viewCObjects get created. One is created when you send the open message in your viewSetupFormScript; the other as a result of the normal view instantiation process. The symptom you see in your program is that you close X and it's still there. You can even look at X in the Listener and see that its viewCObject is nil, but there is an extra viewCObject floating around, still being drawn. Unfortunately, the view system doesn't check for this anomaly during view instantiation and handle it more gracefully.

There are two pieces of advice that you should follow which avoid this problem entirely. First, if a view can be opened naturally, let it be opened. Don't explicitly call Open. Second, don't send Open messages, or any other messages, to your children until your viewSetupDoneScript executes. Your children are not created until your viewSetupDoneScript executes.

Declaring Your Base View

Conceptually, your base view is declared to the root view automatically after your package is downloaded and your InstallScript runs. It's name is your AppSymbol. That's why `GetRoot().AppSymbol` returns your base view. In NTK, you set your application's AppSymbol in the Settings... dialog.

The preceding comment is qualified with "conceptually" because your base view isn't installed in the root view by the declare mechanism I've already described. The end result is the same–your base view is preallocated in the root view–but the process isn't the same (you'll note your base view's template doesn't have a preallocatedContext slot).

Declaring Linked Subviews

The linked subviews feature of NTK adds a twist to declaring views. The best way to think of a linked subview is as a place holder and a linked-layout. You drag the place holder (the LinkedSubview icon from the proto palette) to the parent you want your view to actually have. Then you use the Link Layout... command to link it to a layout. The place holder is just that, a place holder. The linked layout is substituted for the place holder when your project is compiled. All traces of the place holder are lost, except...

If you want to declare a linked subview, you should name and declare just the place holder. You do not need to name or declare the linked-layout. In fact, you are unable to declare the linked-layout because its Allow Access popup menu is empty–it appears to have no parents. You may want to name the linked-layout so that its children can be declared to it. There is no need to give it the same name you gave the link, but it's a good idea. Figure 5 illustrates these points.

Declare ren to baseView and declare child1 and child2 to stimpy. You are unable to declare stimpy to anyone because it appears to have no parents. This is not a problem–you really want to declare ren anyway. When you declare child1 and child2, the only place to declare them is stimpy; baseView is not in the popup because stimpy appears to have no parents. This is a limitation of using linked views. If baseView needs to access these children you can use code references like `ren.child1`.

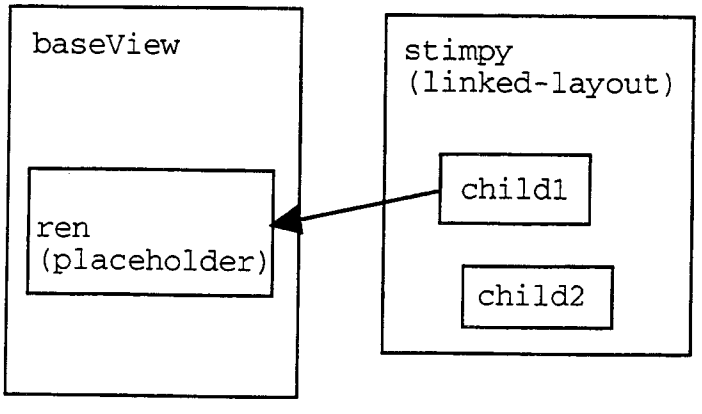


Figure 5 - A place holder and linked-layout relationship.

Note that the names chosen for the place holder and the linked-layout (ren and stimpy) are not the same, but they could be. The symbol stimpy is not available at runtime. If child1 or child2 need to refer to their parent, they should use ren. This illustrates why it's a good idea to name your place holder and linked layout identically–the code is clearer. It's kind of strange for child1 and child2 to refer to their parent as ren, a symbol defined outside the linked-layout in which they're defined.

Dynamically Adding Views

Adding views dynamically actually turns out to be a complex issue. I describe several techniques in the following sections. You need to pick the one most appropriate to your situation.

Don't Create Views Dynamically

The canonical case is a dialog box. You should create the dialog ahead of time (probably using the linked view feature of NTK) and declare it to your base view. Then at runtime open and close it as you please.This is the preferred method–it's by far the simplest and least error prone.

Don't Create Views Dynamically–Yourself

Perhaps you need to create a group of subviews, but you don't know the exact number until runtime. For example, your application may need to compute the number of subviews based on screen size (to maintain compatibility with future products). The best way to accomplish this is to compute your stepChildren array in your viewSetupChildrenScript. Just put in the templates you want and the view system handles creating the actual views.

If you're going to adjust the size of your stepChildren array using AddArraySlot it had better be in RAM. You might need to do:

```
if not HasSlot(self, 'stepChildren) then
  self.stepChildren := Clone(self.stepChildren)
```

HasSlot is used to test if the viewChildren slot already exists in RAM in the view itself. If the viewChildren already exist, you don't need to do anything. Using `self.stepchildren` in the condition instead of HasSlot doesn't work because the _proto chain is followed into ROM. All the explicit selfs are needed lest you accidentally get the stepChildren slot of your parent.

If you want to modify the contents of the stepChildren array after your view is created, and create new views, use the RedoChildren method. First make the changes you want to the stepChildren array, then send your view a RedoChildren method. All of the view's current children are closed and removed, the viewSetupChildrenScript is called, and then a new set of children is created from the stepChildren array. Note that reordering stepChildren and then calling RedoChildren is a way to re-order your child views dynamically.

If You Insist, There's AddStepView

If you really want to create a template at runtime and then make a view from it, use the global function AddStepView. It takes two arguments: the parent view you want your view added to and the template to use in making the view. It returns the view that is created. Be sure to save this return value so you can access the view later.

AddStepView doesn't force a redraw. You need to either:

- Send the new view a Dirty message;
- Send the view's parent a Dirty message. This is useful if you're adding multiple views–one Dirty and they all show up at once (if you're adding several views, you should also consider using the RedoChildren technique); or
- If the template has the visible bit cleared, just send the new view a Show message. This is useful if you want the view to appear with some sort of special effect.

In addition to creating the view, AddStepView also adds its template to the stepChildren array. This means that the stepChildren array needs to be in RAM so it can be expanded or AddStepView will fail. See the previous comments on ensuring your stepChildren array is in RAM. Do not use AddStepView in your viewSetupFormScript or your viewSetupChildrenScript–it doesn't work. Instead, append your template to the stepChildren array and let the view system do the rest of the work. There is another function, AddView, which works like AddStepView, except it adds the template to the viewChildren array. You should use AddStepView, since the viewChildren slot is reserved for system use.

There is also a pair of functions, RemoveStepView and RemoveView, that are designed to undo the effect of AddStepView and AddView. These functions take one argument, the view (not the template), and remove that view. Unfortunately, they do not remove the corresponding template from the stepChildren or viewChildren slot. You have to do this yourself. If you do not take care of this, your stepChildren slot builds up all these old templates which waste RAM and which may surprise you if something forces a RedoChildren (all the "removed" views' ghosts suddenly come back to life).

BuildContext

Another function that is occasionally useful is BuildContext. It takes one argument, a template. It makes a view and returns it. The view's parent is the root view. The template is not added to any viewChildren or stepChildren array. Basically, you get a free-agent view.

Normally, you shouldn't need BuildContext. It's useful when you need to create a view from code that isn't part of an application (when there's no base view to use as a parent). For instance, if your InstallScript or RemoveScript needs to prompt the user with a dialog, you need to use BuildContext to create the dialog.

Creating Templates

All but the first of the techniques presented require you to create templates. You can do this using NewtonScript to define a frame. You have to remember whether the oneLineOnly bit gets set in the viewFlags slot or in the viewJustify slot. It's easy to mess up.

I recommend creating a user proto in NTK and then using it as a template. That allows you to take advantage of NTK's slot editors. At runtime you can access the user proto as PT_<filname>. If there are slots whose value you can't compute ahead of time, leave them out of the user proto, and at runtime create a frame with those slots set properly and proto off the user proto.

A typical example might be computing the bounds on the fly. If you define all the static slots in a user proto in a file called dynoTemplate, you can create the template you need using code like:

```
template := { viewBounds: RelBounds (
              x, y, wid, hgt),
             _proto: PT_dynoTemplate }
```

This really shows off the advantage of a prototype-based object system. You create a small object on the fly and use inheritance to get the rest of the values you need. Your template is only a two slot view in RAM. The user proto sits on the card with the rest of your application. The conventional RAM-wasting alternative is :

```
template := Clone (PT_dynoTemplate);
template.viewBounds := RelBounds (x, y,
     wid, hgt);
```

You should also note that for creating views arranged in a table, there is a function called LayoutTable which calculates all the bounds. It returns an array of templates. See the *Newton Programmer's Guide* for details.

Summary of Dynamic View Advice

- Avoid adding views dynamically; use declare and invisible views instead. Barring that, add your templates to the stepChildren array in your viewSetUpChildrenScript.
- Do not send messages to or set slots in templates. This is most commonly done in code that uses the stepChildren array instead of :ChildViewFrames(). It also happens when people use the template passed to AddStepView instead of the view it returns.
- The stepChildren array must be in RAM before you can modify it. Remember, "never try to write to ROM–it wastes your time and annoys the ROM."
- Do not anger AddView.
- Use NTK to define as much of your template as possible.

π

Table 2 - The template vs. view quiz answers.

		template	view
1)	created by NTK	X	
2)	only reside in RAM		X
3)	handles `:Hide()`, `:Open()`, and `:Close()`messages		X
4)	reside in stepChildren and viewChildren arrays	X	
5)	code modifies their slots at runtime - preferably, using `SetValue`		X
6)	use system protos in their proto slots	X	
7)	passed to AddStepView	X	
8)	usually contains few slots (inherits, most of its slots)		X
9)	usually reside in ROM	X	
10)	returned by `:ChildViewFrames()`		X
11)	returned by Debug()		X
12)	is a frame	X	X
13)	has a parent		X

Internet Q & A

Robert E. Bruce
JHU - Kennedy Krieger Inst.
robbruce@jhuvms.hcf.jhu.edu

CyberSpace Surfing

This article is selected threads culled from the comp.sys.newton.programmer Usenet news group. Specifc paragraphs have been left intact, although we have stripped out redundant response content. The complete transcript of this material is available on the PIE Developers source code disk (see the inside front cover).

Soups

From: potts@oit.itd.umich.edu (Paul R. Potts)
Subject: Soup queries failing - why?

OK, to get everyone's mind off non-programming questions and get you chewing on some NewtonScript code, here's a real development question. This is a cc of a message I sent to Kent Borg, but I thought I'd throw the the question out to the wider audience too.
-Paul-

There is a weird bug with my Strainer application. Maybe I'll hold off on sending you the source, since it is really ugly and I don't want to look like too much of a programming dolt. I can describe the problem pretty precisely though.

I played with the inspector for a while last night (docs definitely are incorrect in a lot of places). Here's my best guess currently as to what is happening:

- I'm building a soup, and filling it with items. I can look at the soup anytime and the items are there. So, they aren't actually falling out the other side like it appeared.

-What is happening is this: after a given number of queries, the queries start failing. So, I'll query for an entry, find it, query for a few more entries, then query for the same entry I found earlier, and the cursor will come back nil. The parameters of my query are exactly the same, I can see it in the soup if I dump the soup, it just isn't being found. I've tried resetting the cursor with reset(), and doing a "find all" before narrowing the find. I've got a couple of theories:

1. when you do a lot of finds, sometimes the cursor gets into a state where you have to explicitly set it back so it is looking at the right thing. (Possibly something I can work around with a better understanding of how cursors work); also, I don't understand why it would work right over and over, with :entry() always showing me the right thing, then stop working). Any ideas?

2. there is a memory leak or other problem when querying that leads to queries eventually starting to silently fail. (seems unlikely, since I'm sure soups were really well exercised, but every once in a while in my adventures in coding I do find out that my code is right, it's the system, compiler, libraries, or something else that's wrong)

Here's how I'm doing my query. My app, Strainer, browses three soups: Calendar, Repeat Meetings, and Calendar Notes. The user walks through the entries in one of these soups doing Next(), Prev(), etc. This works great. I've got slots in my app called self.theSoup, which refers to the soup I'm browsing, self.theCursor, and self.theEntry. These hold the current cursor and entry (of course). As the user walks back and forth through the soup that all gets updated, and whatever slots I can interpret get shown on the screen. That's basically how Strainer works.

I've got another soup called toDelete, which is a list of references to entries in the soup the user is browsing. This soup gets created when the user starts browsing, and is automatically deleted when they quit Strainer. Here's how that works:
As the user browses the Calendar soup, for example, they can check or uncheck checkboxes which mark entries for deletion. When the user turns on the check mark, I add an entry to my toDelete soup, and when they uncheck the check mark, I remove it. The entry consists of:

```
// frame: {sourceSoupName: text, entryID: integer, _uniqueID: integer}
// where I don't put the _uniqueID slot in, the system does. The
// sourceSoupName slot refers to the name of the soup the user is
// browsing - for example, Calendar - while entryID refers to the
// _uniqueID field of the entry in its soup. This is enough info so
// that I can later look through my toDelete soup and get an entry,
// and delete it from the soup that holds it.
```

So far, everything appears to work fine. The Calendar soup gets browsed, and entries get added to and deleted from my toDelete soup just the way I'd expect.

toDeleteSoup and LocalCurse are slots in my app that hold references to the soup of entries to delete and a cursor on the soup. If, when the user gets tired or browsing the Calendar soup, they want to strain out all the entries they have marked for deletion, they choose "Strain," tell the app that yes, they really want to do this, and then I walk through the toDelete soup, look up each entry in its appropriate soup, and delete it. This seems to work great.

The problem comes in when the user has checked a bunch of entries in the Calendar soup, thus adding entries to the toDeleteSoup that refer to them. Every time the user moves between entries in the Calendar soup that they are browsing, I query the toDelete soup to determine if the entry is checked or unchecked for deletion - that is, if the entry has a corresponding entry in my toDelete soup, I show the checkbox checked, if not, I show the empty check box. Here's how I'm doing my

query. (If I print out the parameters of my query, they always appear to be correct).

```
LocalCurse := query (self.toDeleteSoup, {type: 'index,
    validTest: func(sEntry)
        HasSlot(sEntry, 'sourceSoupName) and
        HasSlot(sEntry, 'entryID) and
        self.theSoup.theName = sEntry.sourceSoupName and
        self.theEntry._uniqueID = sEntry.entryID} );

// this find seems to work great a number of times, and then starts
// failing, even though the entry I'm looking for is in the soup,
// and all the vars look OK in the inspector when I print them.
// I've tried comparing their text representation instead using
// SPrintObject(sEntry.sourceSoupName) but that doesn't help.

// An example would be: there are ten entries in my soup. I find
// these ten in order, then maybe skip around and do ten or twelve
// more successful finds, then the find starts failing (returning NIL),
// even though if I dump the soup I can see the entry there, right
// where I left it, and self.theSoup.theName is correct, as is
// self.theEntry._uniqueID. LocalCurse:Entry() just starts returning
// nil. Interestingly, the query always seems to continue working for
// one item in the Calendar soup - the last item I added.
```

(I know my method for working with the soup is terribly inefficient and slow; I could use an array of delete entries, or cache the status of the check mark so as not to do a search each time. I wanted to try it this way, though, because it seemed like a great way to exercise the soup functions and really get to know how to use them).

I'm going to post this to the programmer group too... I'm really baffled. If anyone at DTS is interested in looking at it, thank them in advance for me...

-Paul-

From: steved@sdohrmann (Steve Dohrmann)
Subject: Re: Soup queries failing - why?

This may not be the source of your problem but in your validTest you have:

```
self.theSoup.theName = sEntry.sourceSoupName
```

I believe this is comparing object references, not object contents. I think you should be using:

```
StrEqual(self.theSoup.theName, sEntry.sourceSoupName)
```

If you did not clone the string when filling the entry slot, i.e.:

```
myNewEntry.sourceSoupName := theSoup.theName
```

then it's possible that the reference compare opertion your using is evaluating True for those cases. Good luck.

Steve Dohrmann

From: Joe Shmoe <JShmoe@engin.umich.edu>
Subject: Re: Soup queries failing - why?

Thanks, I have actually found my bug. In an earlier incarnation of Strainer I realized that the pointer comparison of strings above could be causing the problem. The problem was, I then changed my code to do content comparison as above, and it *still* failed, so I figured that wasn!t the problem.

It turns out it was the problem; there was *another* bug in my code which was generating the exact same symptoms. With two bugs generating identical symptoms, I would fix one, test it, remove the fix because it didn!t work, then fix the second bug, test it, and remove that fix because *it* didn!t work. Aaarrrggghhh!!! The solution came to me in a trance at about 2:00 in the morning when I said to myself, OK, there is another bug, and then set about proving my hypothesis. (Shades of Zen and the Art of Motorcycle Maintenance...)

For those interested in the gory details: the bug above was in fact a faulty comparison of strings; the trouble is, it seemed to be working part of the time, which really confused me. You would think that a pointer comparison would fail always unless the pointers were really the same object, but somehow it looked like it was coming out right for a

The other bug was that when I added an item to the soup, I wasn!t sure if all the cursors on that soup would be updated, so in order to try to fix the first bug, I was searching the soup for the item after it was added. This had the side effect of resetting the pointer to the first entry, which was causing the lookup failure. Since I was working on one part of my code and largely unaware (I had forgotten) that another part of my code was changing slots behind my back, I didn!t notice this for a long time.

Things in the NTK that could possibly have helped diagnose this: first of all, does it really make sense that the default comparison for strings is pointer comparison? I mean, it makes sense for all kinds of other objects, for space reasons, but I can!t think of a lot of situations where you would be comparing strings and want to actually be comparing pointers. This might be a good candidate for an optional warning generated by the compiler (Warning: pointer string comparison (do you really want to do that?))

Another thing that would help would be a way to set a breakpoint to go off if the value of a slot changed. This is how I would have done it with Visual BASIC or C++.

Well, anyway, it was my bug. I!m pleased to say that all my test code designed to see if I could flog the soup manager to death behaved perfectly well. Soup handling really is some phenomenal code.

-Paul-

There is a weird bug with my Strainer application.

From: potts@oit.itd.umich.edu (Paul R. Potts)
Subject: more soup details

regarding my soup problem... it is getting curioser and curiouser. One clue might be: if I do a gc() after each find, the Newton will then only find the *last* entry I added to the toDelete soup, no previous ones. All the entries are still clearly there in the soup where I left them, if I dump the soup. It appears the gc() is blowing away all the old cursors, except that then the Newton tries to recycle them, perhaps? and fails. I spent some time last night talking to a PIE engineer and he suggested many things to try; I've managed to try most of them, like doing MapCursor with my test function; it fails, too, after a certain number of iterations; printing out the values that I'm comparing to inside ValidTest, which appear fine, etc. Still no clue.

I'm trying to come up with a simple piece of code to run in the inspector which illustrates the problem I've having. In the meantime, take a look at the following in the inspector:

st:=getStores()[0]<returns the internal store>

cal:=st:GetSoup("Calendar") <returns the calendar soup>

a:= array(0,nil) <returns an empty array)

```
while cur:Entry() do begin AddArraySlot(a,
cur:Entry()._uniqueID);
   cur:Next() end
```

<select multiple lines and hit enter to run this. This adds the contents of the _uniqueID slot of each calendar soup entry to the array>

a <dumps the array>

```
foreach index, value in a do
 begin
   if cur:Entry() then print ("found match")
 else
   print ("testing")
```

<does nothing! Inspector doesn't return anything! why not?>
<even the following minimal example:>

```
foreach index, value in a do begin print (index); print(value)
end
```

<also does nothing in inspector. why not?>

I've gotten foreach to work fine in code; cant' figure out why it doesn't do anything in the inspector window. I'd like to use the expression above that tests for hits and put it in a function, then execute the function repeatedly until it starts to show the failure (no entries found that match).

-Paul-

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: more soup details

The inspector window is a funny beast, don't assume that it's part of the root view (blindly send messages, you need to access root using Getroot(), and then send the message).

—Kent

From: Joe Shmoe <JShmoe@engin.umich.edu>
Subject: Re: more soup details

How would I apply this? Send foreach() as a function call to the root view returned by GetRoot()? Or did I accidentally declare my vars to be in the root view!s context? If that!s the case, I!d expect some sort of exception when I tried to access them. Can you give an example of how I could use a foreach() call from the inspector window?

Thanks,
-Paul-

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: more soup details

12 second hack:

```
call func()
begin
  local foo := {x: 1, y: "ha", z: 42};
  foreach slot,value in foo do
    print(slot && value);
end
with ()

"x 1"
"y ha"
"z 42"
#2      NIL
```

From: waboring@apple.com. (Bud)
Subject: soups and deleting apps

I was reading up on stores and soups a little bit last night, and a question came to mind. What happens to an apps soup if the app is deleted from the newton. Does the o.s. remove the soup, or does it stay around forever?

Walt Boring

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Re: soups and deleting apps

It stays, that is why someone created the souper utility. It calls the systems functions to remove the soups.

regarding my soup problem... it is getting curioser and curiouser.

From: potts@oit.itd.umich.edu (Paul R. Potts)
Subject: Re: soups and deleting apps

Has anyone seen the following symptom with Souper: it doesn't display the last soup? For example, mine doesn't show the To Do soup; if I run a little app of mine that makes a soup, it then shows the To Do soup but not my soup.
-Paul-

From: scleaves@kepler.unh.edu (Stephan R Cleaves)
Subject: Re: soups and deleting apps

You can probalby remove your soup in the RemoveScript of your application using the soup:RemoveFromStore() function. This would ensure that you don't leave memory sitting around tied up with old junk.

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Soups and such.

Like Bill, I am uncertain about the proper way to code soups in the Newton. I think I am going to have an application which would benefit from being able to just extend the name soup by one or two slots. Like what if I want to add social security number as a slot for everyone. Is this possible? It seems to be but I don't know what command I use to add it to the soup. Do I just redefine the name soup and add the slots which I need?

Thanks
Robert Bruce

From: chris@cad.gatech.edu (Chris McClellen)
Subject: Re: Soups and such.

All you need to do is load an entry from the soup, and add a slot to that entry. a Soup is just a collection of objects. To add an object, just do : SoupObj:Add(), to add a frame to a soup. I suppose you can equate that to adding a "slot" to a soup, but in my mind, you just add an object (frame) to the collection.

Here's an example:

Suppose soup "MySoup" exists in main memory

```
Soup := getStores()[0]:GetSoup("MySoup");
Soup:Add(.....);
```

and thats all there is to adding to a soup. the "....." should be a frame of yours that you wish to add.

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Re: Soups and such.

I'm posting this message I got from Mike Engber.

The key is Union Soups. The docs cover all sorts of soup functions, but the basic 3 calls that should handle most of your needs are:

```
GetUnionSoup(<soup name>);
<union soup>:AddToDefaultStore(<new entry>);
EntryChange(<entry>);
```

At the bottom of the message is code snippet showing how an app should create it's own soups.

> I think I am going to have an application which would benefit
> from being able to just extend the name soup by one or two slots. Like
> what if I want to add social security number as a slot for everyone. Is
> this possible? It seems to be but I don't know what command I use to add
> it to the soup. Do I just redefine the name soup and add the slots which
> I need?

You can add slots to individual entries in the Names soup. Here's a simple example you can try from the Inspector. It will add a field to the first entry in the card file soup.

```
soup := GetUnionSoup(ROM_CardFileSoupName);
curs := Query(soup,{type: 'index});
entry := curs:Entry();
entry.ssn := "123-45-6789"
EntryChange(ROM_CardFileSoupName);
```

In general, apps are encouraged to use existing soups as much as possible. To avoid making a big mess, an app should create a single slot, using its AppSymbol as the slot name - then you can put a frame of whatever data you like in that slot.

So the 4th line in the above example would be more properly written:

```
entry.|crazyEx:PIEDTS| := {ssn: "123-45-6789", otherData: 42};
```

In a separate post I'll put some information on the recommended way to choose unique AppSymbols, etc.

-ME
Copyright: © Apple Computer, Inc. 1993 author: Mike Engber
SoupCreation.f v1.0b3

Snippet for an app handling soup creation:

- respects user's preferences as to where data should go
- don't have to worry about what stores are available
- handles thorny case of user swithching cards while your app is running

```
//4 base view slots
appSymbol:  '|foo:PIEDTS|,
appObject:    ["widget","widgets"],
soupName:   "fooSoup",
soupIndicies: [{structure: 'slot, path: 'fooIndex, type: 'string}],

//2 base view methods

//call from your viewSetUpFormScript
//returns a union-soup for your app to use
InitSoup: func()
 begin
    CreateAppSoup(soupName,soupIndicies,
        EnsureInternal([appSymbol]),
        EnsureIternal(appObject));
    AddArraySlot(CardSoups,soupName);
    AddArraySlot(CardSoups,soupIndicies);
    SetUpCardSoups();
    GetUnionSoup(soupName);
 end,
```

```
//call from your viewQuitScript
DeInitSoup: func()
 begin
  SetRemove(CardSoups,soupName);
  SetRemove(CardSoups,soupIndicies);
 end,
```

Cryptic Notes:

Whenever a card is inserted, the OS makes sure that every soup listed in the CardsSoups global variable exists on the card, creating soups as necessary.

CardSoups: a global variable (array) whose elements are used pairwise. Each pair is a soupname (string) and an array of indicies (to be used in creating that soup).

CreateAppSoup: creates the soup in the internal store (if necessary)

SetUpCardSoups: Makes sure every soup in CardSoups is on the external store(s). This fn is normally called by the OS whenever a card is inserted. InitSoup() calls it to handle any cards that have previously been inserted.

The base view slots, appSymbol and appObject, are standard slots and are used by the OS for other things (e.g. filing, routing, ...). appSymbol is the symbol your app used to declare itself to the root view (set in the Preference dialog in NTK - default is KitchenSym). appObject is an array of two strings which describe the singular and plural of the type of data your app deals with. This snippet makes the assumption that the soup being created corresponds to that data.

The slots soupName and SoupIndicies are just names I picked for this sample code. Their main purpose is to tuck away the soup's name and indicies so I can pass the exact same object to both AddArraySlot and SetRemove (recall, SetRemove uses pointer equality to find the element to remove)

From: tonym@netcom.com (Tony Mann)
Subject: Re: Soups and such.

Robert Bruce (robbruce@jhuvms.hcf.jhu.edu) wrote:

: So the 4th line in the above example would be more properly written:

: entry.|crazyEx:PIEDTS| := {ssn: "123-45-6789", otherData: 42};

If you want to create an index on your new slot, then this method will not work. Instead, begin the slot name with the initials of your app, so at least the odds of a name conflict are lessened. Moreover, someone examining the soup will know where the slot came from.

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: How to search a soup for items in a folder

I am wanting to create an array from the Name soup. I have the loop working which steps through the soup and grabs the first and last name of everyone in the soup. I now want to have the program search the soup for people in the business folder and then compile list.

The search on page 15-17 of the manual is doing this kind of search but the code is supposed to be used with the folder header. If I don't want to use the folder picker is there any way to automatically tell it to search for things in a particular folder.

Robert Bruce

From: kentborg@world.std.com (Kent Borg)
Subject: Re: How to search a soup for items in a folder

If you know the name of the folder, sure, stick that in the query. Now how to get a list of all the names without using the folder header is another matter. It is too late for me to know the answer.

—
Kent Borg

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Re: How to search a soup for items in a folder

What is the syntax of the query?

```
nameSoup:=GetUnionSoup("Names");
cursor:=Query(nameSoup,{type:'index,validtest: name.labels = "Personal"});
```

I don't think that's right.

Robert Bruce

From: yandell@mullauna.cs.mu.OZ.AU (Peter YANDELL)
Subject: Re: How to search a soup for items in a folder

```
cursor := Query (nameSoup, {type:'index, validTest:
        func (entry) entry.labels= 'Personal});
```

Pete.

Sounds

From: ses@tipper.oit.unc.edu (Simon E Spero)
Subject: Where are the system sounds

Can anybody how I can get references to the system sounds? I had a trawl through the system soup, but I could find where the actual sounds lived.

Thanks
Simon

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Where are the system sounds

Remember this thing, don't always assume that a particular sound resource is available in every single future NEWTON ROM, this might or might not be true. We at DTS are just now trying to figure out what sounds should be officially blessed, in other words they will stay around forever and ever. The best thing is to include your own sounds so you are sure they are part of the resources.

Meanwhile, here's some code Mike Engber in our DTS group wrote to poke around looking for interesting sounds, it has some other intersting things for you hackers to think about :-).
—
Here's some code you can use in the inspector to search for magic ptrs to sounds. You should easily be able to generalize it for other purposes.

```
call func()
begin
 for i := 0 to 369 do
  begin
   local mp := call compile("@" & i) with ();

   if ClassOf(mp) = 'Frame AND mp.sndFrameType then
    begin
     write("PlaySoundSync(@");
     write(i);
     write(");");
     write($\n);
    end
  end;

end with ()

PlaySoundSync(@4);
PlaySoundSync(@16);
PlaySoundSync(@51);
PlaySoundSync(@62);
PlaySoundSync(@76);
PlaySoundSync(@77);
PlaySoundSync(@85);
```

Writing Text

From: bharat@cs.stanford.edu (Bharat Shyam)
Subject: Writing out Text

I have tried writing out Text using the clParagraphView without any success. It does not seem to change the displayed text when I change the Text slot in the template corresponding to this view has anyone succeeded in displaying text and changing the displayed text when the user does something (such as tapping on the screen.

Note that I want to display the text in the same view without actually drawing another view. I want to simply change the text displayed in a view at runtim.

Thanks Much
Bharat

From: Joe Shmoe <JShmoe@engin.umich.edu>
Subject: Re: Writing out Text

I haven!t tried with with a clParagraphView, but for other kinds of views, if you change them, send the view a :Dirty() message. This will cause it to get redrawn. I!ve got a view for example that contains several static text fields; the enclosing view gets a :Dirty message after I change the text and they redraw. You could experiment with sending it to the child view that actually changed or the parent view. Hope this helps...
-Paul-

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Writing out Text

SetValue will automatically call dirty for views that need updating, such as text slots in clParagraphViews. SetValue is you buddy.

—Kent

Disappearing Views

From: hinds@cmgm.stanford.edu (Alexander Hinds)
Subject: Problems with disappearing views

I have a program in which I add views dynamically. To do so, in the viewSetupDone script, I have

self.viewChildren := Clone(viewChildren);

Later on,

AddView(aProtoAppView, someViewTemplate);

Everything works fine when I run the program once. However, upon a second invocation, any system defined children views (of the protoApp) are never displayed! (ie. the close box, the border, etc...)

I have localized this behaviour to the line with the Clone() call. When removed, everything works fine.

Anybody have any ideas? Thanks in advance. I'm baffled.

—X

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Problems with disappearing views

ProtApp consists of a clView, a protoStatus and a protoTitle, in other words three different protos, each one has its own childView array, and yes, this is inside the ROM, so if you want to mess with this you have to clone it.

However, there's another runtime based array called the stepViewArray, and during runtime this one contains all the valid views (ChildViewFrames will return valid frames from this entity).

Use AddStepView if you want to add things into this view, if possible avoid the viewChildren array (AddView).

This is documented in a nice Q&A and in future documentation. We are looking at ways at distributing the current batch of Q&A documents, they are up on AppleLink at this point of time, and all NTK owners will have access to the archives.

—Kent

From: hinds@cmgm.stanford.edu (Alexander Hinds)
Subject: Re: Problems with disappearing views

Well, I tried this, and it still behaves as before. This is really frustrating. Any views I add in the NTK or at runtime behave normally. Those added by the protoApp still disappear upon the second invocation of the program.

—X

Pen Tracking

From: dejohnso%asylum.cs.utah.edu@cs.utah.edu (David Johnson)
Subject: Help with pen tracking

I can't seem to figure out what to use to track the pen. I see clickScript, but that is only for first contact. I see stroke, but that is only at completion. I see how to get all the points in the stroke (PointArray or something?), but again, that only seems to be at the end of the stroke. What do I use to detect any movement of the pen?

Also, I think I am missing something big with the Inspector. I don't seem to be able to use it for debugging, it just prints out messages, but I can't query it. Any suggestions?

Thanks
David Johnson

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Help with pen tracking

Most likely you are not using Enter when you enter executable code. If you have multi-line statements, you need to select these and then hit enter.

Finally, remember, the inspector environment is not tied to the root view, so you really need to write function blocks in many cases in order to get something done.

—Kent

Dial()

From: eskin@btree.uucp (Michael Eskin)
Subject: Dial() not implemented?

In the sound section of the NTK docs there is a Dial() function that doesn't seem to be implemented. Am I missing something?

Michael Eskin
Newbie NTK programmer

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: Re: Dial() not implemented?

Try:
GetRoot().Dial("5551212",
userconfiguration.currentDialSpeaker);

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Dial() not implemented?

From a Q&A about Sound available on ALink (and I hope all NTK owners should have access some time next week to the archives):

The Dial() Function

Q: I want to generate a unique tone when users press the number keys on my calculator (a cute user-option). But I cannot get the Dial() function to work. I tried using the Dial function within my scripts, and get an error that seems to indicate that the function doesn't exist. I also tried Dial in the Inspector with the same results.

A: The Dial function is actually a rootView message, so if you are inside the view system, you need to precede a call to Dial with a colon. Dial takes two parameters: a string that specifies the number to be dialed, and a parameter that defines where the tone should be played (internal speaker is the default.)

Ex:
GetRoot():Dial("1234", userconfiguration.currentDialSpeaker);

—Kent

clPictureView

From: freeman-eric@cs.yale.edu (Eric Freeman)
Subject: clPictureView

Has anyone had luck including a bitmap in a clPictureView? I can't figure how to include the bitmap after I'm in the browser editing the icon attribute.

Thanks,
Eric

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: Re: clPictureView

The picture must exist as a PICT resource and you must include that resource file in your project. Once you've done that, adding a slot called "icon" and selecting the "picture" editor will show you a list of resource files in your project and. for the selected file, the names of all the pictures contained within. This means that you must give names to your PICT resources within the resource file.

Another alternative is to use the Project Data file to load the resource into a NewtonScript variable that you can then stuff in a slot. Here's an example of code you might put in the Project Data file:

```
// open the resource file
// (change this to the FULL path to your resource file)
r := OpenResFileX("Macintosh HD:Newton:picture.rsrc");

// Read a picture resource and stuff it in a variable
// The first arg is the resource name, the second controls whether a
// mask is created or not
```

```
myPict := GetPictAsBits("Sample", true);

// Close the resource file
CloseResFileX(r);
```

Now all you have to do is create a slot, set the editor to "evaluate" and set its value to myPict. Then you can say something like:

```
self.icon := :parent().myPictSlot;
```

Obviously, the real use for something like the Project Data example above is to make large data structures of image and/or sound info (see the KidCards source for info on including sounds in your project). For example, you can make an array of images and assign that array to a single slot, accessing the pictures by index rather than having a bunch of separately named slots.

Scrolling Windows

From: drarick@panther.weeg.uiowa.edu (David Rarick)
Subject: How do you make windows scroll?

I have some text that fills up at least five screens. How do I make them scroll up & down, and overview (belly) buttons work?
—
David A. Rarick

From: tonym@netcom.com (Tony Mann)
Subject: Re: How do you make windows scroll?

Respond to the viewScrollXXX messages, and either use SetOrigin or SyncScroll.

Linked Subview

From: drarick@panther.weeg.uiowa.edu (David Rarick)
Subject: Bug in NTK with LinkedSubview???

I think I found a bug in the NTK (either that, or I'm doing something wrong...) When I link a LinkedSubview to another layout, if I try to remove the LinkedSubview and link another one to the same layout, NTK says that the layout is already linked. I have tried removing the linked layout from the project, and then adding it again. The only way I can get around it is to select all, create a new document, paste all into the new one, and then save it with the same name, replacing the original.
—
David A. Rarick

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Re: Bug in NTK with LinkedSubview???

I have had this problem as well. I don't know any other work around other than the recreation one you suggested. Misery loves company don't ya know. My project folder is strewn with variations on the same name for the same layout.

From: tonym@netcom.com (Tony Mann)
Subject: Re: Bug in NTK with LinkedSubview???

Fortunately there is an easy way around this. Here is the official solution:

This is a known problem with NTK 1.0b4, however you can workaround it.

>Open the offending layout file with ResEdit and remove the FMST resource.
>NOTE: removing other resources is downright dangerous.

I wish I had known this when I was going crazy trying to unlink views! :-)

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: Bug in NTK with LinkedSubview???

If I remember correctly the release notes warned about relinking linked views to other layout views. Yes, this should be fixed in later beta versions.

NewtonScript Syntax

From: wkearney@access.digex.net (William Kearney)
Subject: Syntax in newton script?

I could use a little help in clarifying a newton script question. I want to use protoLabelInputLine with the labelCommands variable. I don't want to keep the array of values in the view's labelCommands variable, but in the parent. The main reason for this is the array of values is going to be used by multiple protolabelinputlines and will be dynamic. Eventually, the array values will be stored in a soup for strorage.

Now, if I create a slot that contains the array in the parent, what should the labelCommands slot in a child view of this parent, contain to reference it?

To further illustrate:

```
protoApp:Sample
    Test:Evaluate("["ABC", "XYZ", "123"])
_protoLabelInputLine
    labelCommands:Evaluate("["ABC", "XYZ", "123"])
```

Works fine, the picker pops up with the items in the array, but does not use the array Test from the parent. How should I code the labelCommands to use the Test array?

As in:

```
protoA
pp:Sample
    Test:Evaluate("["ABC", "XYZ", "123"])
_protoLabelInputLine
    labelCommands:Evaluate("["ABC", "XYZ", "123"])

                    ^^^^^^^^^^^^^^^^^^^^
What goes here? ———————— ??
```

Thanks!
Bill Kearney

From: Bob Ebert <ebert@newton.apple.com>
Subject: Re: Syntax in newton script?

Normal NewtonScript inheritance should take care of this. When looking for the value of a slot, first the local context is checked, then the global context, then the slots in the "self" frame, then the slots in the proto chain, then the parent frame, then the slots in the parent's proto chain, and so on.

So, if there's a labelCommands slot in the parent, it should be referenceable from children. In your case, you simply wouldn't add a lableCommands slot in the protoLabelInputLine.

That said, there are syntaxes in NewtonScript that can be used to avoid checking the proto and parent chains. (i.e. self.slotname won't use the parent chain.) So, if the protoLabelInputLine is implemented using this syntax, inheritance won't work.

Also, if anyone in the proto chain has a slot called labelCommands, that will stop the inheritance. In this case, the protoLabelInputLine prototype has a default value for labelCommands... nil. So the inheritance stops here.

So how do you do what you're trying to do? Recall that when you assign an array (or any non-immediate) to a slot, no copy of the array gets main. So, in the viewSetupDoneScript for your input line, you could write a line of code like:

```
self.labelCommands := :Parent().labelCommands
```

This should make the child's labelCommands slot refer to the same array in memory as the parent's labelCommands slot. (i.e. if you add or remove array elements via the parent's pointer, the child's array will change to.)

—Bob

Child View Slots

From: Mat Havoc <havoc@uchicago.edu>
Subject: Setting Slots in Child Views

I may have missed something really basic here, but can anyone tell me if there is a simple and efficient way to set the slots in a child view from a method running in the parent view? I have tried registering the child view in the parent and then passing the registered name to SetValue(), however I just get an error saying NewtonScript expected a frame (and presumably did not find one). I also tried just referencing the slot with the "." operator, to no avail. I can see the obvious solution of creating a method in the child to set its own slots and invoking it when I want to update the data, however it seems horribly inefficient to spew out a method every time I need to tweek a single integer's value. Any suggestions?.

From: potts@oit.itd.umich.edu (Paul R. Potts)
Subject: Re: Setting Slots in Child Views

Well, I don't know if this is efficient, but I'm putting links to my child views on the parent view, declaring them, linking them, and then using them as paths to the child object. For example, if my parent is main, I declare a linked subview called subLink, link it to the child form, and then from the context of main do self.subLink.object. You can send messages to the linked view too, like self.subLink:Open(). Make sure you give the views names and allow them to access their parents. -Paul-

---

From: kaas@newton.knoware.nl (Peter Kaas)
Subject: Re: Setting Slots in Child Views

Hmmmm... :)

Suppose you have an application view with a child view. By selecting the child view in NTK, and pressing command-i (get info) you are able to give the child view a name. Also click "allow access from ..." with the name of the application view (first, make sure the application -has- a name).

Say, we named the child "ImportantView". To set slot "ImportantData" in thechild view from within the application view:

```
func()
    begin
        self.ImportantView.ImportantData :=
        { When:"Fast", Direction:"Pizza Bar"};
    end;
```

or use (i'm not totally sure about this one since I haven't got an index in my version of the manual, so i couldn't check out the syntax):

```
func()
    begin
        SetValue(self.ImportantView,'ImportantData,
        { When:"The Year 2400" });
    end;
```

Hopla..., correct me if i'm wrong :-)
—
Peter Kaas.

PS: Could somebody tell me how to get rid of that really *** geneva 9 font
in the script editor in NTK?

Comm Help

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: Renewed plea for comm help

I don't want to sound like a broken record, but for a personal communicator, the Newton seems awfully impersonal. I have tried every permutation of the information in ch. 20 of the NTK manual in an attempt to get the Newton to make so much as a peep out the serial port, and *nothing* seems to work. I know I'm not stupid, so it must be the Newton's fault! ;)

Seriously, here's the latest code hack. I've given up on the protoSerialEndpoint and in desperation am looking at ADSP. In my application view, I have a slot called "theEnd", where:

theEnd := {
I also have a slot called "theEndAddr" where:
theEndAddr := {

The addressData slot refers to a version of ClarisWorks listening forn ADSP connection, using the ADSP Tool with the appropriate NBP name registered (i.e. cshotton:ADSP@UT Backbone).

Executing the following code: results in an untimely death somewhere within the Instantiate method. At least this code doesn't lock up the Newton as it would if "theEnd" was a protoSerialEndpoint, so I guess I can count this as progress.

My frustration level at this point is about as high as it's ever been when confronted with a programming challenge. The simple task of getting the Newton to talk to another computer is obviously achievable, since the stupid Toolkit package does it. Communications is the MAJOR reason I bought the Newton.

I hate to sound paranoid, but a lot of the new commercial software coming "real soon now" is certainly able to communicate using all of the IR, ADSP, serial, and modem capabilities. I have to wonder if the communications functions aren't perhaps purposefully absent from the NTK in order to protect the initial market share of these products from cheap or free versions of the same software developed by less mercenary developers.

I also can't understand why, if these functions obviously exist and work in Apple's Toolkit package and these commercial apps, Apple can't release a single, tiny, functional snippet of code showing how to do this (unless the previous paragraph is true...)

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: Re: Renewed plea for comm help (oops)

Ouch. Obviously the code examples that were copied from the NTK file didn't make it through the paste operation in NewsWatcher. Lemme try this again....

theEnd is a slot in the application view:

```
theEnd := {
  _proto:protoADSPEndpoint,
}
```

theEndAddr is another slot in the application view:

```
theEndAddr := {
  optionType : 'address,
  optionLavel : kCMARouteLabel,
  opCode     : opSetRequired,
  data : {
    addressType: kNamedAppleTalkAddress,
    addressData: "cshotton:ADSP@UT Backbone"
  }
};
```

Calling the following:

```
func()
begin
  theEnd:Instantiate(theEnd, nil);
  theEnd:Connect(theEndAddr, nil);
  theEnd:Disconnect();
  theEnd:Dispose();
end
```

causes the Newton to cough up a -48214 error in the middle of the Instantiate method somewhere. I'll FedEx a case of beer to ANYONE who can supply some functioning communications source code for the Newton that allows it talk to something besides itself (or another Newton).

(insert the rest of the tantrum from the previous message here.)

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: An answer to => Renewed plea for comm help

Kent Sandvik asked me to repost the following e-mail message from him here.

The error would indicate that you are trying to modify a read-only object, theEnd object, a quick hack is to clone this one so it's a real object in memory, or create the endpoint in RAM, instead of creating the endpoint in the slot, do something like this in a viewSetupDoneScript or any other useful system message:

theEnd := {_proto: protoADSPEndpoint};

It's always good to create endpoints in RAM, this because you want to sometimes modify them 'live', and read-only ROM objects are thus bad.

From: sandvik@newton.apple.com (Kent Sandvik)
Subject: Re: An answer to => Renewed plea for comm help

We know of the current comms development issues and problems, especially the issue of lack of samples. The push is to have something available mid next week, including modem, serial port and ADSP code. Stay tuned for more info, I will try to work my butt off to get this done :-).
—Kent

Button Renaming

From: robbruce@jhuvms.hcf.jhu.edu (Robert Bruce)
Subject: Renaming a button

I want to have a button labelled "start" change to "stop" once you click on it. I have the program altering the button.text slot correctly. If I print(button.text); before and after it is changing the text. However I can't get the button to refresh. I have tried to use the button:dirty(); command which I use successfully to update a static text box with no avail. Are buttons different in this repect? Is there a proto for a button which changes text when you click on it?

Thanks
Robert Bruce

From: agc@mercury.mercury.uah.ualberta.ca (Andrei Chichak)
Subject: Re: Renaming a button

Use setvalue() instead. A simple assignment sets the value of the memory region but does not update the screen. Setvalue() also marks the page dirty.

From: wag@netcom.com (Will Wagner)
Subject: Re: Renaming a button

I don't know if this will work, but I would try calling the SynchView method and set up the text in the vewSetupFormScript method. If that didn't work, or instead of, I would try SetValue(button, 'text, "My Name") to see if that works. If that doesn't work, I guess I would use masking tape over that part of the screen and write it with a ballpoint pen :)

Will Wagner

From: steved@sdohrmann (Steve Dohrmann)
Subject: Re: Renaming a button

I had the same problem with some other views. Try:

```
SetValue(myButton, 'text, "New Text");
```

This should work. As far as I can tell,

```
myView.text := "New Text";
myView:Dirty();
```

is not equivalent to using SetValue. One caveat: SetValue will call the view's viewChangedScript method so if you are using that hook already you may need to differentiate between changes due to user action (e.g. writing) and programmatic changes via SetValue. Good Luck.

- Steve Dohrmann

Project Data

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Newsgroups: comp.sys.newton.programmer

Does anyone have any information on what the Project Data choice under the Project menu in NTK is for? I would guess that it's for project-specific data, much along the lines of the GlobalData file in the NTK folder. But functions, frames, etc. that I define here are not able to be referenced from methods in my application.

Page 3-10 of the NTK Guide provides one sentence: "Project Data: Opens a Project Data text editor window, where you enter application data in a special data format."

From: Robert Bruce <robbruce@jhuvms.hcf.jhu.edu>
Subject: Re: Project Data in NTK

It allows you to compile additional functions before your app is compiled if I understand things correctly (which hasn't been frequently lately). Check out the Kids Card demo for a suitable use of the Project Data file.

Robert Bruce

From: cshotton@oac.hsc.uth.tmc.edu (Chuck Shotton)
Subject: Re: Project Data in NTK

KidsCard is an "interesting" starting point. As best I can tell, stuff in the Project Data file is interpreted at compile time and used to load external resources, scripts, etc. into NewtonScript symbols that the compiler can merge with the "normal" code generated from the view editor. Where are all of these "calls" documented? (e.g., OpenResFileX, GetSndAsSamplesRate11KHz, etc.) They don't appear to be in the current NTK docs.

(Munging through the NTK application with ResEdit and Norton Utilities turned up all sorts of other goodies, too. If you're terminally curious, it's worth a peek!)

Chuck Shotton

From: Bob Ebert <ebert@newton.apple.com>
Subject: Re: Project Data in NTK

The Project Data file is just a text file that gets interpreted by the NTK compiler when your project is compiled. It's the first thing that gets evaluated, so any objects that are created as a result are available to reference in the rest of your application.

There are also two special objects that you can create, called InstallScript and RemoveScript. NTK will take these two scripts from Project Data and use them for your apps InstallScript and RemoveScript. Nothing else from Project Data is saved into your application uness it's referenced somewhere by the rest of the app.

—Bob

User Interface

From: kentborg@world.std.com (Kent Borg)
Subject: Kent Borg's User Interface Note #1: Save Your State

Kent Borg's User Interface Note #1: Save Your State

Newton is not Macintosh. On a Macintosh applications are launched and quit. On a Newton applications are opened and closed. There is a difference. For one, on the Newton there is no "process menu". If I have several applications open at once I can't go over to the righthand edge of a menu bar and list the "running" applications. Newtons don't do that.

The answer is to close the application that is in the way. The way to operate on a Newton is to open things you are interested in and close things that you are not interested in. Cool.

The only catch is that we shouprogrammers ld not distroy stuff just because the user wanted to make or refer to a note "down there at the bottom" on the Notepad.

When you close your view, save your state.

I just downloaded a bunch of applications from the BCS Mac BBS (where the Newton stuff is locally). I also have the built in applications. Which ones do it right?

The Good Ones. Close and reopen these and nothing changes:

- Notepad Though being on the bottom, the Notepad gets it for free.
- Names
- Inbox
- Outbox
- Connection
- Sharp
- Card
- Styles
- Timezones
- Prefs
- 800#s One of the first Newton "Books". Remembers the page. 5 bonus points.
- Days Another of the first Newton "Books". Remembers the page. 5 bonus points.
- Hot Buttons A mini "Extras" drawer, remembering the buttons is the essence of this utility, so no points there. However the position of its little pallette *is* remembered. Nice touch. 3 bonus points.
- Settings Allows the Newton's map location to be set to a place not in Apple's atlas. Again, remembering is the essence of this utility, so no special credit here.

The Nearly Good:

- Calculator On open the last value is the same as it was on close, but all the previous history is lost.
- Dates It preserves whether you are looking at the To Do or Calendar view, but all other "view" information is lost.

The "Too" Good:

- Find It remembers the last find even if I highlight something else before asking to find. It should have the text I highlighted waiting in the find blank.

The Bad:

- Formulas Particularly the currency exchange rate should be kept: am I supposed to keep writing the exchange rate in over and over? Stupid.
- Mine
- Tapboard
- Minefield "I was in the middle of a game when the phone rang...I had to make a note and I lost my game!" Game boards should only clear when the player *says* so.
- ObjViewer Because this is a hacker's tool, I forgive it a few crimes...but if there is no theoretical reason why it can't remember too...
- Tricorder Those *really* important notes I might write in the one edit field... :-)
- Birthdays Once I wait for this application to mention that Aunt Minnie's is next week, why toss that information?
- Resto
- Check Please These two are utilities for helping split up restaurant bills. In both cases, after I have worked hard entering numbers, why lose my work just because I want to use my Newton for something else?
- Moon Tells phases of the moon. Doesn't remember where I left off.
- Assist Particularly when the user has navigated to a particular "How Do I?" screen, a new "Assist" without any highlighted text should bring the user back to that same text—I think...you might talk me out of this one, you might confront me with some user testing or something unfair like that.

Conclusion:

The imbedded Apple applications do a pretty good job of not trashing the user's context, but *none* of the freeware/shareware applications consider the user's context—except the two Newton books, and the ones where the configuration is the very point of their existance.

Please Save the State!

—

Kent Borg

π

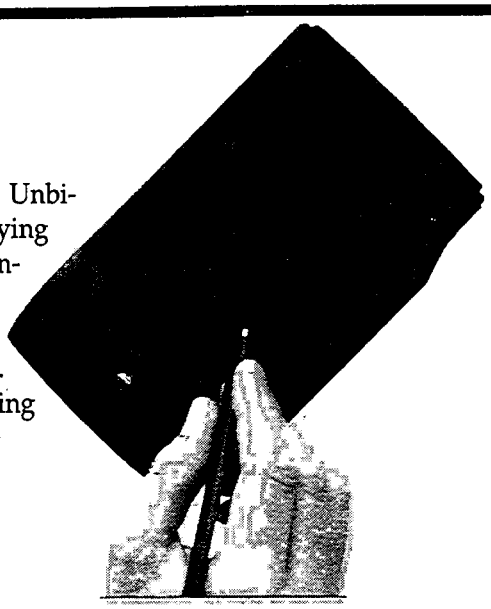
Robert Bruce is the current Moderator of the Newton archive at bnnrc-srv.med.jhu.edu and the Newton section of the Sumex, Info-mac archive. He is also producing an arcade style game for the Newton. Look for it soon on an internet site near you.

Newton is not Macintosh.

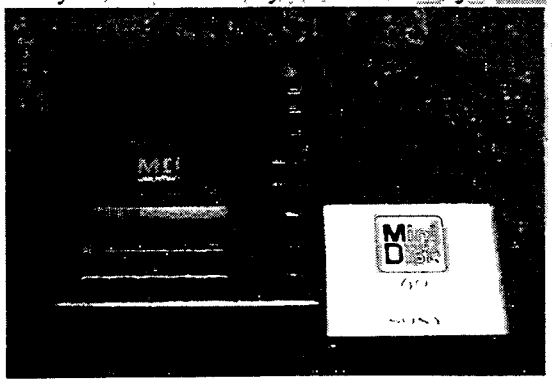
Portability!

Covering The Portable Consumer Electronics & Computer Market

Ever Wonder Why There Is So Much Confusion?



Informative articles will help you when making that portable purchase. Unbiased stories and reviews will come in handy when you are not buying something based on heavy opinionated views. Enlightening and entertaining details on using your portable electronic in all sorts of environments and conditions. A focus on new technologies - like MiniDisc and Personal Digital Assistants will keep you current on the latest portables. Don't delay. Get your subscription to PORTABILITY! and stop getting confused by others. Send in your check today!



Want Something You Can Understand?
Read PORTABILITY!

Choose From The Options Below* **Renew ?** ☐

- ☐ One (1) year. $20 - Six (6) issues (U.S.)
- ☐ Two (2) years. $35 - 12 issues
- ☐ One (1) year. $40 - Six (6) issues (OVERSEAS)
- ☐ Two (2) years. $75
- ☐ One (1) year. $30 - Six (6) issues (Canada/Mexico)
- ☐ Two (2) years. $55

Fill in your Name, Address, City, State and Zip Code. PLEASE PRINT!

NAME _____

ADDRESS/BUSINESS _____

CITY/ ST/ZIP _____

COUNTRY/PROVINCE _____

*U.S. Funds - check, money order, cashier's check

ACT NOW!

SEND TO:
Perfection Applied
P.O. Box 1783
Orem, UT 84059
or call (801)222-0120 for more info

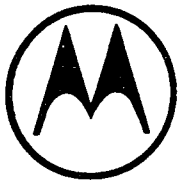
Think Newton.™
Now think wireless.
Next call Motorola.

Motorola has announced plans to develop a device with Newton-Intelligence that promises *anytime, anywhere* wireless communications.

If your customers can benefit from the freedom to communicate without tying themselves to a phone line, we'd like to talk to you now about our upcoming developer programs.

No obligations...no commitments up front. Let's just start the dialogue. Let us know what you're doing and how we can help you in your efforts to go wireless.

Start by sending an AppleLink to MOTONEWT.DEV (Internet: MOTONEWT.DEV@APPLELINK.APPLE.COM) —our Developer Team is waiting to hear from you!



MOTOROLA

Get Wired!

" Wired looks like Vanity Fair should, reads like Esquire used to and talks as if it's on intimate terms with the power behind the greatest technological advance since the Industrial Revolution. "

David Morgan, Reuters

WIRED

Subscribe!

1 Year subscription (12 issues): $39.95
That's 33% off the newsstand price.
Call: 1-800-SO WIRED
Email: subscriptions@wired.com